MAX/MSP/JITTER for MUSIC

A PRACTICAL GUIDE TO DEVELOPING **Interactive Music Systems** FOR EDUCATION AND MORE

V. J. Manzo

Max/MSP/Jitter for Music

This page intentionally left blank

Max/MSP/Jitter for Music

A Practical Guide to Developing Interactive Music Systems for Education and More

V. J. Manzo

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Oxford University Press, Inc., publishes works that further
Oxford University's objective of excellence
in research, scholarship, and education.

Oxford New York
Auckland Cape Town Dar es Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in
Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

Copyright © 2011 by Oxford University Press

Published by Oxford University Press, Inc.
198 Madison Avenue, New York, New York 10016

www.oup.com

Oxford is a registered trademark of Oxford University Press

All rights reserved. No part of this publication may be reproduced,
stored in a retrieval system, or transmitted, in any form or by any means,
electronic, mechanical, photocopying, recording, or otherwise,
without the prior permission of Oxford University Press.

Cloth: 9780199777679
Paperback: 9780199777686

1 3 5 7 9 8 6 4 2

Printed in the United States of America
on acid-free paper

Contents

Foreword by David Elliott • xi

Preface • xiii

Acknowledgments • xix

About the Companion Website • xxi

1 Introduction to Programming • 1
Covers the basic tools of operation and navigation in Max as well as an introduction to the basic ways to work with data.
- Introduction to Max • 3
- The Max Window • 7
- Help Patchers • 8
- Arguments • 9
- Separating Items in a Message • 10
- Numbers: integers and floating points • 10
- Aligning • 12
- Commenting • 13
- Inspector • 13

2 Generating Music • 17
Introduces elements of algorithmic composition. We will create a program that randomly generates pitches at a specified tempo. The program will have the ability to change a number of musical variables including timbre, velocity, and tempo. We will also write a program that allows your MIDI keyboard to function as a synthesizer. These two programs will be the basis of future projects related to composition and performance.
- The RAT Patch • 17
- MIDI (Musical Instrument Digital Interface) • 18
- Synthesizing MIDI numbers • 19
- Adding Timing • 21
- Slider Patch • 28
- Rat Patch 2 • 29
- MIDI Input • 32

3 Math and Music • 38
Discusses some of the math used in musical operations. We will examine the math behind musical concepts like transposing music by some interval and adding chord tones to a root note. We will also look at some of the things that will help make your program look better and more accessible to users. We will create a program that harmonizes MIDI notes.
- Math in Max • 38
- Forming Intervals • 43
- Window Dressing • 45
- Creating Chords • 48
- Presentation Mode • 51
- Further Customization • 52
- Comments • 56

4 Scales and Chords • 58
Teaches you how to build scales and chords and play them back in a variety of ways. By the end of the chapter, you will write a program that allows users to play diatonic chords in a specified key using just the number keys on your computer keyboard.
- Scale Maker • 58
- Chord Maker • 61
- Order of Operations • 70
- Setting Values • 74

5 Interactive Ear Training • 82
Teaches you to make an interactive ear-training program.
- Ear Trainer • 82
- Further Customization • 92

6 Data Structures • 95
Teaches you to write a program that randomly generates diatonic pitches at a specified tempo. We will learn how to filter chromatic notes to those of a specific mode by using stored data about scales. By the end of this chapter, you will have created a program that composes diatonic music with a simple rhythm. We will also learn about adding objects in order to expand the Max language.
- Adding External Objects • 95
- Tonality in Max • 96
- Filtering Chromatic Notes to Diatonic Notes • 97
- Random Tonal Music • 99
- Abstractions and Subpatchers • 103
- Working with Pitch Classes • 105
- Tables • 111

7 Control Interfaces • 119
Examines some premade patches demonstrating a few techniques for designing diatonic musical instruments. We will review some of the basic ins and outs of MIDI, learn some ways to program more efficiently, and discuss a number of control options for your patches.
- bpatchers • 120
- Saving Settings • 125
- Storing Presets • 137

8 Control Interfaces Continued • 140
Looks at some innovative ways to control music making as we develop musical instruments. We will use your computer keyboard and mouse as performance instruments as well as discuss the use of videogame controllers in your patches.
- Arguments for Abstractions • 141
- Using the Mouse • 144
- Using Videogame Controllers • 151

9 Tools for Music Theory Concepts • 157
Allows you to design some tools to aid in the discussions of concepts related to music theory. In particular, we will discuss chord progressions, scale analysis, chord analysis, mode relationships, harmonic direction of chords, and harmonization. By the end of this chapter, you will have an arsenal of tools for explaining theoretical concepts of music.
- Chord Progressions • 157
- Scale Analysis • 160
- Mode Relationships • 167
- Harmonic Direction • 170
- Harmonization • 172

10 Working with Time • 178
Discusses aspects of time, rhythm, and the sequencing of events. You will learn to create interactive performance and compositions systems as well as create patches that demonstrate rhythmic complexity. By the end of the chapter, you will have created patches that can record and loop MIDI sequences as well as a number of patches that work with notes over time.
- Sequencing • 178
- Step Sequencers • 189
- The Transport • 194
- Overdrive • 198

11 Building Stand-alone Applications • 200
Analyzes a "Chord Namer" application that allows a user to enter a chord name and see the notes on a MIDI keyboard. We will "build" this patch as a stand-alone program that can be used on any computer even if it does not have Max installed.
- Preparing the Application • 200
- Building the Application • 206
- Icons • 209
- Permission and Cross-platform Building • 209

12 Working with Audio • 212
Discusses MSP, a collection of objects that work with audio signals. The MSP objects can handle actual sound recordings, like audio from a microphone, as well generate signals.
- Basic Ins and Outs of Audio • 212
- Sine Waves • 218
- Timbre • 221
- Synthesizer • 226
- Synth Building • 228

13 Audio Playback and Pitch Tracking • 237
Looks at some of the ways that you can play back and record sound files. We will also look at some ways to track the pitch of analog audio and convert it into MIDI numbers.
- Playback • 237
- Pitch to MIDI Tracking • 243

14 Audio Buffers • 250
Discusses how to record audio into a storage container called a buffer. By the end of this chapter, you will be able to record a performance through your microphone and loop the recording while you make sonic changes to it. You will also learn how to make a polyphonic synthesizer that uses a single recording of your voice as the pitches.
- All about Buffers • 250
- Recording into a Buffer • 256
- Referencing Playback Speed to a MIDI Note • 258

15 Audio Effects and Processing • 263
Addresses implementing audio effects into patches with live audio and sound files.
- Preparing the Patch • 264
- Delays • 266

- EQ • 271
- Other Effects • 275

16 Working with Live Video • 277
Discusses *Jitter,* a set of objects that handle video and visual-related content in Max. By the end of this chapter, you will have created a patch that changes the color of a live video when pitches are played.
- Matrix • 277
- Camera Input • 278
- Adjusting Color • 280
- Mapping MIDI Pitches to Color • 284

17 Working with Video Files • 290
Works with preexisting video files located within the Max search path. By the end of this chapter, you will have created patches that detect presence in certain areas of a video. We will also examine some aspects of tracking colors in video.
- Video • 290
- Presence Detection • 292
- Color Tracking • 301
- Preassembled Video Patches • 303

18 Video Research Instrument • 305
Looks at a research instrument designed to measure the time between stimulus and response when a participant is watching videos.
- Stimulus Testing Instrument • 305
- Working with Paths • 309
- Application • 312

19 Informal Music Learning Instruments • 314
Looks at a collection of interactive systems used in a research project involving students who have not had any formal music training; the goal was to teach music composition and performance to these students using software instruments that allow them to play chord functions with ease.
- E001: keys mapped to chords • 314
- E005: pitch tracking to chords • 316
- E003: form and progressions • 317
- E002: number-based composition • 320

20 Compositions and Perception Tools • 325
Examines some ways to interact with audio processing objects. We will examine compositions for live acoustic instruments and Max. The remaining example patches in this chapter will deal with audio processing as it relates to hearing and some aspects of perception.
- Composition for E♭ Clarinet and Computer • 325
- Hearing • 329
- Perception • 330
- Conclusion • 332

Foreword

Max/MSP/Jitter for Music—aptly subtitled *A Practical Guide to Developing Interactive Music Systems for Education and More*—is a groundbreaking, step-by-step approach to empowering the creativity of music educators and music students through interactive computer software design.

V. J. Manzo provides a brilliantly organized, detailed, and illustrated explanation of how to create, customize, and individualize music software to support whatever music teachers and students want to do: compose, perform, improvise, analyze, research, and/or enhance and assess student's development of musical concepts, aural skills, notational and theoretical competencies, knowledge of multiple musical styles, and much, much more.

Divided into a careful sequence of twenty chapters, this practical guide begins with a clear introduction to the nature and uses of *Max/MSP/Jitter*, or *Max* (for short). *Max* is distinctive among music programming languages because it employs a user-friendly graphic interface that facilitates both formal and informal music teaching and learning. Importantly, this book is not a dry handbook/textbook; it includes a wealth of practical examples and tutorials, sometimes in easy-to-follow "cookbook" format.

Among the themes and strategies discussed and implemented in *Max/MSP/Jitter for Music* are discussions of: generating music through algorithmic composition; the "math" underlying musical concepts, theory, and harmony; and ways of building scales and chords in "painless" ways for use in personalized creativity. Among many other highlights of the book is the way Manzo explains how to create musical instruments with *Max*; build and visualize chords on a MIDI keyboard; record and play back student performances and compositions; and use *Jitter* to create and manage video content.

V.J. writes in a very clear and personal style, which is based not only his vast knowledge of music software and hardware, but on his many years of practical experience as an extraordinary classroom teacher. Indeed, I have seen V. J. in action on numerous occasions. He embodies all the attributes of the complete musician-educator: artistry, creativity, savvy, intelligence, and highly informed action.

David Elliott
February 2011

This page intentionally left blank

Preface

An interactive music system is a hardware and/or software configuration that allows an individual to accomplish a musical task, typically in real time, through some interaction. Though commonly associated with composition and performance, the tasks associated with interactive music systems can include analysis, instruction, assessment, rehearsal, research, therapy, synthesis, and more. These systems typically have some set of controls, hardware or software, such as switches, keys, buttons, and sensors by which musical elements like harmony, rhythm, dynamics, and timbre can be manipulated in real time through user interaction.

In this book, we use the programming language Max/MSP/Jitter to write custom software for musical interaction. We discuss the concepts needed to complete your project, complete many projects in a step-by-step style guide, and look at examples of working systems. Emphasis is placed on the pedagogical implications of software creation to accomplish these tasks. Whether you want to create a program for composers that explores relationships between two modes or an exercise for beginners that helps improve finger dexterity, you will soon learn how writing customized software can supplement and complement your instructional objectives. We also discuss ways to interact with the software beyond just the mouse and keyboard through use of camera tracking, pitch tracking, videogame controllers, sensors, mobile devices, and more.

Why Design Custom Software?

Today, there are software applications for just about everything, but to what extent do we allow music software to dictate how we teach musical concepts? After installing a software application, it's normal to look at the program and ask "what does it do," "how can I perform with this," and "how can I make a demonstration or instructional activity out of this for my class?" There's certainly nothing wrong with this, but you may already have some musical ideas in mind and are looking for a way to express them using the efficiency and interactivity of technology. However, existing software may not be able to address the particular concepts you want to address from the angle you prefer.

Imagine teaching harmony with the aid of a specialized program that showed common tones between the chords and scales, or a program that used the first seven number keys to play the seven diatonic chords of a key. Imagine

composing a piece of music with a program that showed how chord functions tend to resolve in a given key.

Software developers typically design a program's layout to be accessible and intuitive, but in doing so, they are bound to show certain biases toward the visibility of what are considered the more common features. In an instructional setting, if the feature that is going to help the instructor explain concepts of rhythm or harmony is somewhat buried in the program's menus, he or she may be less inclined to teach those musical concepts right away because there is too much requisite knowledge of the software involved just to get to the desired menu. Instructors would have to teach a number of software concepts just to get to the place where they could teach the musical concept they wanted to address in the first place. It's not the software company's fault; after all, they don't know what and how you teach. However, it's a common case of technology dictating the instruction instead of instruction dictating the use of technology.

This problem is not unique to technology. Even the conventions of traditional notation using staff paper can dictate how we're going to teach; if we don't enjoy counting notes on ledger lines, we just stick to writing notes on the staff. In the same way, it's just as easy for software to confine us. If we want to teach some musical concept in an interactive way using the efficiency of technology but can't find the technology to support it, the notion of an interesting approach to teaching the concept likely gets dropped.

At the same time, teaching with technology can be seen as trendy and gimmicky. Suppose you decide to write a program that plays diatonic chords in a key by using the buttons of a videogame controller. The activity in a classroom setting can be fun, but at the same time, it can be pretty pointless if the program doesn't address some musical concept and the activity isn't accompanied by solid teaching. However, if these things are in place, the student is then able to accomplish some musical task using a controller that is easy to use—and probably more familiar than that one-octave xylophone he's hated using all year. It's easy to worship technology because of its "ooh wow" factor, especially in a classroom setting. However, after the novelty of the technology wears off, we're still music educators first and technologists second. A good interactive system should allow a user to do musical things with efficiency, greater control, and clarity; it should not just exist for the sake of having technology in the classroom.

Technology in the Classroom

Having an arsenal of customized software to explain specific musical concepts can make your teaching life so much easier. How many times do you really need to write out the whole step and half step patterns of a major scale on the chalkboard? What if you had an application that showed the pitches, and steps for a scale, any scale, starting on any pitch? Not only will using such an app save you

time, but it's a program that you could allow your students to download and interact with on their own.

Many teachers are terrified of teaching "technology lessons" because they don't want to be in the situation where some little kid knows more about technology than they do, when the reality is that having a kid like that in class can be an incredible benefit! That kid already understands the technology, so first, he'll be the cheapest and most accessible tech support you'll ever find when something goes wrong in the classroom—let him troubleshoot for you and his peers when things stop working. Second, and more important, since he already understands the *technological* side of things, it helps you to conceptually explain to him the *musical* side of things, which I guarantee he doesn't understand nearly as well as you. Musical concepts, as you'll recall, are the sorts of things you went to school to learn about.

These days, you can't walk into a convenience store without seeing seven or eight computers being actively used for a specific intended outcome—why should the music classroom be any different? Even band classrooms! Technology truly can help facilitate our teaching objects. However, students don't just need teachers to show them new tech toys or cool software; they can find that on YouTube, from their friends, and in their local music store. What they need is for a trained musician to help them make sense of the musical world around them—the use of interactive music systems can make this process richer and more palatable. That's knowledge students can't find just anywhere.

Book Design

This book is divided into 20 chapters that address common uses for the software Max/MSP/Jitter as they pertain to music educators. The book is structured in a step-by-step guide by which we'll build programs. We address the musical concepts and pedagogical implications behind the programs we write and provide instruction for adaptation and expansion of the programs.

If you are an instructor using this book as the text for a semester course, I would recommend covering about two chapters per week with the exception of longer chapters like 6 and 12 or wherever a little more time is needed on a given topic. New material builds upon previously learned concepts, so while skipping chapters is possible, students should be told to read the Help file for any unfamiliar objects or terms. Each chapter concludes by providing a list of objects and terms introduced in the chapter. There are assignments at the end of each chapter that are somewhat more rigid in design at first, but soon allow the reader some level of individuality in completing them. Regular assessments and individualized projects are strongly encouraged.

Basic Troubleshooting

The activities provided in this book have been tested, and you should not encounter any problems—in theory! As you move through the book, you're going to have to act as your own troubleshooting team if something goes awry. For every activity we do, there is a picture reference in the text and a working example of the software available on this book's companion website (see "About the Companion Website" following the acknowledgments).

A few basic steps will help to make sure that you don't run into problems:

1. Save early and save often—ensure that you save your work frequently and that you keep your saved files organized. Don't accidentally overwrite files with the same name unless instructed to do so in the text.
2. Follow the steps—ensure that you've followed the instructions exactly as written by connecting things correctly and entering text correctly, noting spaces if present.
3. Consult the picture and reference patch—as mentioned, a working example of each activity is available from the companion website.
4. Restart the program—if you are still running into problems, save your work, restart the program completely and re-open your work.
5. Restart the computer—if your computer is being unresponsive, it is a good idea to shut it down, wait for a minute or so, and turn it back on (don't forget to turn it back on as this is considered to be an important step).
6. Getting help—consult the floating Max Window, the Max Help files, and the Max reference files, all of which are explained in the first chapter.
7. If for some reason you still encounter an error, you should try to recreate the error on a different computer to identify problems with your machine; it's always possible that your volume is down, speakers are unplugged, keyboard is not working, and so on. Errors like this are best discovered in the privacy of your own home to minimize the amount of humiliation experienced.
8. Cycling '74, the authors of the software Max/MSP/Jitter have a support forum on their website. However, you should refrain from posting your questions on this site until you have read through this book and all the reference files (including tutorials) in the program. The users on the forum are helpful for matters that go beyond information that is clearly covered in the aforementioned references. If you're a student, it's also not a good

> idea to try to pay people on the forum to complete assignments for you. Users will generally ignore or even laugh at you, and will likely refer you to the reference manuals. If you are reading this book as part of a course, the chances are strong that your instructor subscribes to the forums and will see your post.

This book and the activities described herein are compatible with both Max versions 5 and 6. Small differences between the two versions are noted in the text. As new versions of Max are released, some changes to the program documentation, layout, and functionality may occur that are different from how they appear in this text. Simply consult the Help files and revision history documents provided within the program in these cases. Be patient; some concepts are a little tricky and you may have to reread something a few times before it sinks in. All in all, Max/MSP/Jitter is not hard to learn, but be patient. We'll create software with Max in a way that exposes you to the language, but keep in mind: there's more than one way to write the same program. Soon, you'll get the hang of it and will be writing your own programs in no time. Enjoy!

This page intentionally left blank

Acknowledgments

I would like to thank my colleagues in the John J. Cali School of Music at Montclair State University. The methods in this book were largely refined through my class instruction and could not have been established without your curricular support. Thank you also to the undergraduate and graduate music majors for helping me to refine the way I teach programming concepts to nonprogrammers, and for generously giving your time to test the directions contained in this manuscript. Thank you to Gary Shur and Matt Skouras for meticulously and repeatedly reading through each step and providing such valuable feedback and input.

To my colleagues in the Kean University Conservatory of Music: thank you for helping to shape my implementation of technology as it relates to instruction, composition, and performance. To Matthew Halper, thank you for your friendship and willingness to participate in a perpetual conversation about music theory since 2000.

To my Ph.D. advisor Deborah Sheldon in the Boyer College of Music and Dance at Temple University: thank you for your openness to spending time talking about my research ideas involving technology for music education, and your incredible enthusiasm. To Maurice Wright, Allison Reynolds, Richard Brodhead, Ed Flanagan, and the rest of the music department faculty: thank you for making Temple's PhD program in music education one that fosters and values cutting-edge technology and ties it to practical and effective musical outcomes. This book would not have been possible without your assistance and guidance.

To my colleagues in the Technology Institute for Music Education (TI:ME) and the Association for Technology in Music Instruction (ATMI): thank you for your continued support and input about the use of technology in music instruction. My sincere thanks to Rick Dammers, Jim Frankel, Scott Lipscomb, Bill Bauer, Mark Lochstampfor, Sandi MacLeod, Jenniffer Snodgrass, Jay Dorfman, Will Kuhn, Marj Lopresti, and Steven Kreinberg for your support and motivation.

Thanks to Michael Halper and my colleagues in the Information Technology program at New Jersey Institute of Technology. Thanks also to Scott Skiibo for doing some good work in that area.

To the faculty at New York University, especially Luke Dubois, Kenneth Peacock, Robert Rowe, Joel Chadabe, Juan Bello, and John Gilbert, thank you for demonstrating the vast implications for technology in music. My continued thanks to David Elliot for your support, insight, and collegiality.

Thank you to David Cope, Peter Elsea, and Paul Nauert at University of California, Santa Cruz. Even though Max is a long way from LISP, I continue to apply concepts and strategies I learned from you in many things I do. Divide and conquer!

This book, of course, could not be possible without Cycling '74, the many members of the Cycling '74 forum, and the numerous Max developers and artists throughout the world. Special thank you to Miller Puckette, Brad Garton, Tristan Jehan, Ivica Ico Bukvic, Masayuki Akamatsu, Takeru Kobayashi, Olaf Matthes, and Jean-Michel Couturier for giving me permission to use their objects as part of the EAMIR SDK. Thank you to Jonathan Bailey for your programming expertise and assistance in developing the Modal Object Library. Thank you, also, to all users of the Modal Object Library and EAMIR forums, and to John Deacon, Brian May, Farrokh Bulsara, Roger Taylor, and Roy Thomas Baker for your continued inspiration and encouragement

To Norm Hirschy at Oxford University Press: thank you for reaching out to me about writing this book. My continued thanks to you and the rest of the team for your professionalism and expertise during this process.

Of course, I would be lost without the support of my family, Raquel and her family, and all of my friends. Thank you!

About the Companion Website

www.oup.com/us/maxmspjitter

Oxford has created a password-protected website to accompany *Max/MSP/Jitter for Music*. The site features documented source code for each example in the book. Throughout the text, the caption for each figure pictured includes the filename of a patch that you can open as a reference example. Simply follow along in the book while developing your own software or using the example code in case you get stuck.

The site also features numerous examples of larger projects written in Max/MSP/Jitter, third party external objects and patches addressed in the text, and other media data which you can modify and use for your own projects.

You may access the site using username Music4 and password Book2497.

This page intentionally left blank



Introduction to Programming

A programming language is a means to specify processes performed by a computer. A programming language is somewhat similar to the language we speak. Each word in the language serves some function and the ordering of each word follows a syntax from which we derive meaning.

In a programming language, these words are commonly referred to as *functions* or *objects*. Each object has some purpose germane to the programming language. For example, an object called *number* might allow a user to specify a number. Another object called + might be used to add that *number* to another *number*. The objects, each with a specific purpose, work together to create some intended outcome; this is how a program works. In this case, the program we defined added two numbers together.

In a typical programming language, there are numerous objects, each with a specific function. One difficulty in the beginning stages of learning to program is that you do not know the fundamental objects that form the basic vocabulary of that programming language. In this book, we will begin by developing small music programs that use only a few objects. As we progress in chapters, more objects will be introduced and your programs will become more sophisticated as you incorporate more objects into them.

Another difficulty lies in thinking about the steps involved in creating a program. There are different ways to write programs that achieve the same result, but they all require that precise activity be specified in order to produce that result. In my first experience as a kindergarten music teacher, I stood at the

door of my classroom and greeted the students as they entered the room. "Good morning, let's take out our books and start class." At that moment, they hadn't yet put their backpacks down, walked to their seats, or sat in their chairs. It was obvious that they were not ready to begin class. Some students scrambled to find *any* book anywhere in the room. Some took all of their books out of their backpacks. One student began crying. It was a disaster.

The problem was that I didn't clearly describe the processes that I wanted to take place. Instead of using a number of small processes in conjunction with one another, I attempted to use one command that would do it all. It is important to think of the overall flow of actions before you begin to program; break down the objective into smaller tasks and create a procedure for accomplishing the objective.

In this book, we will be using a programming language called Max/MSP/Jitter, commonly referred to as simply *Max*. Max is different from most programming languages in that it uses a graphical interface to display each object as a small rectangle on the screen. This rectangle is called an object and is the basic unit of function in Max.

Objects typically have inlets on the top of the rectangle for receiving data *from* other objects, and outlets on the bottom for sending data *to* other objects. Data are sent from one object to another by drawing a small line, called a *patch cord*, from the outlet of one object to the inlet of another object. With the objects connected via patch cords, it is easy to see the dataflow between each object in your program.

There is an unlimited number of programs you can write using Max. For the purpose of this book, we will focus on the necessary skills to write programs for composition, performance, and music instruction. Those who may find this book particularly useful are music educators who want to supplement their lessons with interactive instructional tools, develop adaptive instruments to aid in student composition and performance activities, and create measurement tools with which to conduct music education research. There are no prerequisite programming skills required at all and the programs we will write can be useful for children as well as adults.

It would be beneficial to read this book while working through the tutorials included in the Max software application. At the end of each chapter, a few related tutorials are given as supplemental reading materials. As you begin to learn this language and grasp the concepts of programming, you will quickly see how easily you can develop software applications to accomplish your objectives.

Example files for each chapter are available from the Oxford University Press companion website for this book. Portions of these files are pictured in figures throughout each chapter. For your reference, in the description beneath each picture is the filename from which the picture was taken.

The steps involved in creating these patches are listed in a "cookbook" style numbered list. If you get stuck or if something doesn't work correctly, go back and ensure that you have completed each step exactly as described and you'll find yourself back on track.

If you have not already done so, download and install the latest version of Max from www.cycling74.com. This book is compatible with Max versions 5 & 6. You will also need to download the demonstration files that accompany this book from the companion Oxford University Press website http://www.oup.com/us/maxmspjitter for this book or from http://vjmanzo.com/oxford. The demonstration files also contain an installer file called *EAMIR SDK* that you should install prior to reading Chapter 6.

Introduction to Max

As we begin building some programs together, be brave. At times, you may not understand the programming concepts at first and it may feel like you're just blindly following instructions. Continue to follow each step in the programming process and these concepts will soon become clear.

In this chapter, we will look at the basic tools of operation and navigation in the Max programming environment. As we spend time discussing these tools, don't expect to make any music until the next chapter. An understanding of these tools is necessary to use the Max program.

A document in Max is referred to as a *patcher* or, simply, a *patch*. A patch is a blank canvas for programmers to put objects on. In Max 6, the program initially greets you with a Welcome Screen, but, for now, close this window leaving only the Max window behind it open. From within the Max application, click the menu item marked *File* at the top of the screen and select *New Patcher* or use the key command ⌘+*n* (Mac) or *ctrl*+*n* (Windows). To create a new object, press the *n* key. When you do so, a small rectangle appears on the screen allowing you to type in the name of an object in the Max language. You can also create an object by clicking the *New Object* icon located at the bottom of the patch window near the Lock icon. You may hold your mouse over these icons to reveal a label. The *New Object* icon opens the *Object Palette* in Max 5 or the *Object Explorer* in Max 6.

It's time to learn our first object called *button*. Note that the object names are case-sensitive.

1. Create a new object box by pressing the *n* key on your computer keyboard
2. Type the word *button* in the new object box
3. Hit the enter/return key

If you've accidentally clicked away from the new object box without typing in the word *button*, you can double click the object box to allow text to be entered once again. Once the word *button* has been entered, the *button* object appears on the screen as a small circle.

The top of the *button* object has one inlet for receiving data. If you hold your mouse over the inlet on top of the *button* object, a red circle appears around the inlet as well as a message window giving you some description about the data that the inlet can receive. Now, hold your mouse over the outlet on the bottom of the object. A button sends out a message called a *bang* from its outlet.

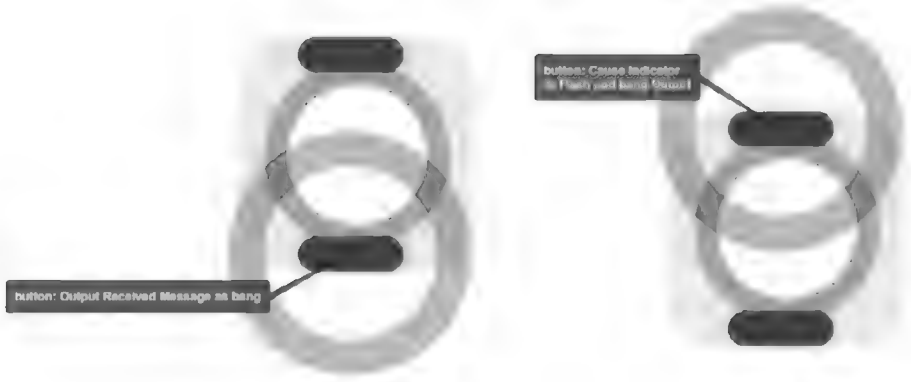


FIGURE 1.1
an inlet circled in red (left) alongside an outlet circled in red (right)

If you are having trouble viewing the inlet, you can *Zoom In* and *Zoom Out* on a patch by clicking *View* from the top menu, or by using the key commands ⌘+= (Mac) or ctrl+= (Windows) and ⌘+- (Mac) or ctrl+- (Windows). In Max 5, you can also click the *Zoom* icon located at the bottom of the patch window.

A bang is an event in Max that triggers an object to perform some task. The only task the *button* object performs is to send and receive bangs. The *button* is such a common object in Max that it has its own key command shortcut, the *b* key, for putting the object in your patch.

4. Add 5 more *button*s to your patch by pressing the *b* key or by highlighting an existing *button* and copying and pasting it. Copying and pasting is accomplished in Max by going to the *Edit* menu or by using the key commands ⌘+*c* (Mac) or ctrl+c (Windows) and ⌘+*v* (Mac) or ctrl+v (Windows). You can also copy an object by holding the alt/option key and dragging an object

To move *buttons* to different locations in your patch, simply click the center of the object and drag it to the desired location. You can also resize the *button* by holding your mouse over the bottom right of the object until your mouse shows two arrows at an angle and dragging while holding the mouse button down.

We've been working in what Max calls the *Patching Mode* or *Patching View*. In Patching Mode, you can create new objects, move them around, resize them, and connect them to each other. By going to the *View* menu at the top of the screen and unchecking the item marked *Edit*, the patch will become *Locked*, that is, the patcher will function as a program and cannot be edited in any way until the patch is unlocked again. You can lock and unlock a patch by clicking the small *Lock* icon at the bottom of the patcher window or by using the key command ⌘+*e* (Mac) or ctrl+*e* (Windows). You can also ⌘+*click* (Mac) or *ctrl+click* (Windows) on any white space in the patch to toggle lock modes.

5. With your patcher locked, click on the *button* objects you've made

Notice that they blink when clicked. This is a graphical display of a bang message being sent or received from the *button* object. Unlock the patch and you can once again make edits in Patching Mode.

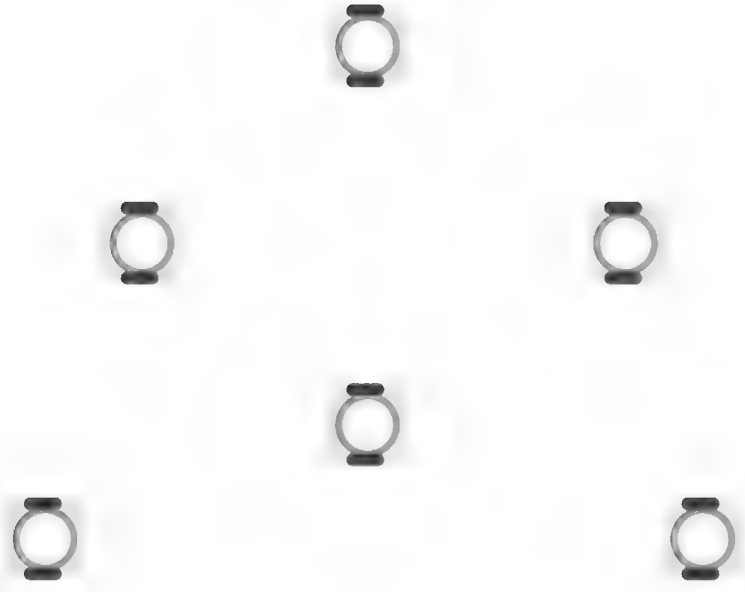


FIGURE 1.2

a group of buttons | button_tree.maxpat

Now let's send data from one *button* to another via *patch cords*. Before we do so, I recommend that you go to the menu item at the top of the screen marked *Preferences* (Mac) or *Options* (Windows) and make sure that *Segmented Patch Cords* is checked. *Segmented Patch Cords* is an option that causes a patch cord to be created when you click on the outlet of an object. The cord will travel wherever your mouse does and will not disappear until you click on the inlet of another object or press the *esc* key. I have found that this option is easier for beginners to work with, so if *Segmented Patch Cords* is not checked, consider clicking on the menu item to enable it. You can always disable it later. To connect objects with the option unchecked, you have to hold your mouse button down from the time you click on an object's outlet until your mouse is over another object's inlet.

Unlock your patch and

6. Click on the outlet of one of your top level *button*s and move your mouse to the inlet of another *button* beneath it until a red circle appears above the second *button*

With *Segmented Patch Cords* selected, the cord will follow your mouse once you click, as opposed to your having to hold the mouse button down while dragging the cord to an inlet. Patch cords always connect from outlets to inlets and never the other way around. Once again, keep in mind that if you click on an object's outlet, the patch cord will follow your mouse until you connect it to another object's inlet or press the esc key.

7. When your mouse gets over the inlet of the second *button*, click your mouse and the 2 objects will be connected

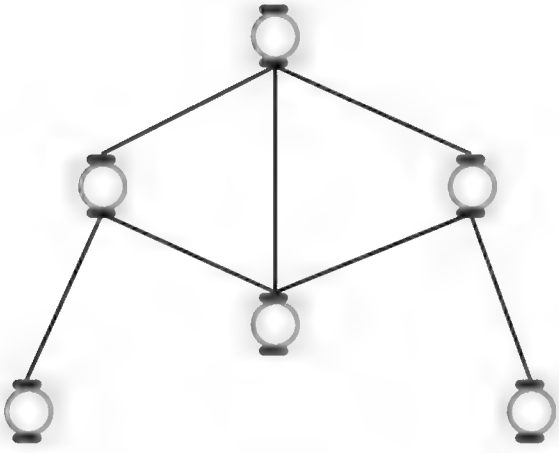


FIGURE 1.3

a button tree | button_tree.maxpat

In this book, when we reference connecting one object's outlet to another object's inlet, this is the process. Please note that Max 6 will display a more rounded patch cable than in Max 5. This can be changed via the *View* menu or the icon in the bottom menu.

8. Lock your patch and click on the top *button* in your *button* tree

The bang flows from the top *button* and triggers a bang to all of the connected *button*s. If you click on a *button* beneath the top one, the top *button* does not turn yellow because it did not receive any data to its inlet. This illustrates the data-flow within Max. To save this patch, click *File>Save* from the top menu or use the key command ⌘+*s* (Mac) or ctrl+*s* (Windows). When asked for a filename, name the file "button_tree." Notice that the Max filename extension *".maxpat"* is given to your file. After you have saved the patch, close it.

Look at the following illustration and try to determine what will happen when you click on one of the *button*s:

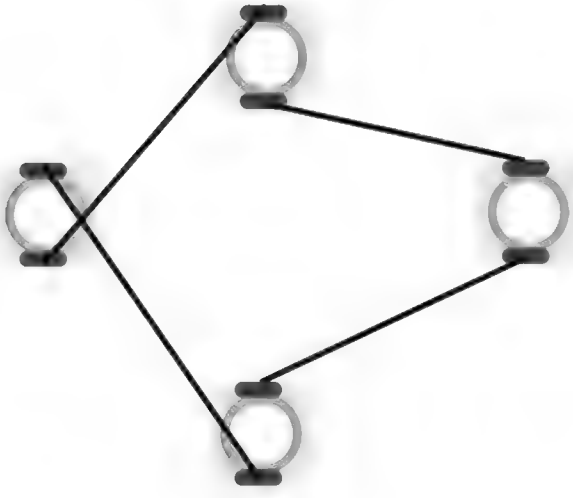


FIGURE 1.4

a button loop

When you click a *button*, a bang will flow from one *button* to another without any way to stop the data. The bang will continue to flow forever without any way of ever stopping. This causes an error called *Stack Overflow*. It's a similar concept to the way that feedback is heard when a microphone is too close to a public address speaker. Make sure that objects are not feeding data back into themselves. Errors like this one will cause an error message to appear at the top of the patch describing the problem and listing the culprit object. Clicking the error message will highlight the problem object in the patch. The error message may be cleared by clicking the small "x" on the right of the message. In some cases, you may be instructed to go to the *Edit* menu and click *Resume*.

The Max Window

Create a new patcher, and

1. Create a new object called *print*

The *print* object is used to display some information in what is called the Max window. Open the Max window by choosing *Window>Max Window* from the top menu or by using the key command ⌘+m (Mac) or ctrl+m (Windows). Important messages, like errors, for instance, will appear in the Max window. If an error occurs somewhere in your patch, you can locate where the error occurred by double clicking on the error message (highlighted in red) in the Max window. This will cause the troublesome object or objects to become temporarily highlighted green.

2. Create a *button* object
3. Connect the outlet of the *button* to the inlet of the *print* object
4. Lock the patch and click the *button*

You will notice that the message "bang" appears in the Max window. The *print* object will print to the Max window any message you send it. Since a *button* sends out a bang message, the *print* object will print the message "bang." Unlock your patch and

5. Create a new object called *message*

A *message* object, also called a *message* box, allows you to enter a message as text by typing into it. The message you type can contain just about anything: words, numbers, words and numbers. Unlike *number* objects whose values are temporarily visible in the object but disappear when the patch is closed and reopened later, *message* boxes will retain the messages entered in them. Objects will receive the data in *messages* and interpret it in some way. The *message* is such a common object in Max that it has its own key command shortcut, the *m* key, for putting the object in your patch.

Be careful that you do not accidentally use a *message* box in place of an object box and vice versa. Notice the difference in appearance between the two.

6. Enter the text *Hello World* into the *message* box

If you've already clicked away from the *message* box, double click the object to allow text to be entered once again.

7. Connect the outlet of the *message* box to the inlet of the *print* object
8. Lock the patch and click the *message* box



FIGURE 1.5
a print object receiving data from a message box | hello_world.maxpat

When the *message* box is clicked, the message *Hello World* will appear in the Max window. You can bring the Max window to the front of the screen by double clicking on the print object.

Help Patchers

If you want help seeing how an object works, you can open up the Help file for each object in an unlocked patch by ctrl+clicking[1] (Mac) or right clicking (Windows) on the object and selecting *Open Help* from the contextual menu. You may also open Help by holding the alt/option key down and clicking on an object in an unlocked patcher. Unlock the patch and

9. Open the Help file for the *print* object

In the Help file for an object, a description of the object is given along with fully working examples of how the object may function within a patch.

The interesting thing about Max Help files is that they too can be unlocked and edited just like patches.

10. Unlock the *print* Help file
11. Highlight all of the objects in the Help patch and copy them using the commands ⌘+c (Mac) or ctrl+c (Windows)

You can construct entire Max patches by copying working code from the Help files. To find out more information about an object including all of the available

1. Mac users will have to use ctrl+click if right clicking has not been enabled in *System Preferences* or if their mouse or touchpad does not support right clicking capabilities.

messages you can send to it, click on the *Open Reference* link within the top right corner of an object's Help file.

At the bottom right of each Help file is a small pull-down menu labeled "See Also." This menu shows objects that are somehow related to the object whose Help file you are currently reading. If you select one of these objects from the menu, its Help file will open up. Looking through these related objects is a great way to learn the Max language fast. Also, if you are trying to find an object that operates similarly, but not exactly the same, to an object you know, you may find that object via the "See Also" menu.

In Max 6, the *Object Explorer* can be used to help you find objects by categories or by searching. This explorer, along with other assistance panels, can be made viewable by clicking *Side Bar* from the *View* menu or by clicking the icon in the lower menu of the patcher. At the moment, these categories probably won't mean much to you, but as you continue to learn Max, searching for just the right function for your patch can become easier if you see objects listed by category.

An additional option in Max is to enable *Help in Locked Patchers*. This allows you to open the Help file for an object in a locked patch by holding the alt/option key down and clicking on an object. To enable *Help in Locked Patchers*, go to the menu item at the top of the screen marked *Preferences* (Mac) or *Options>Preferences* (Windows). Under the *Assistance* tab, check *Help in Locked Patchers*. In this book, we will not open Help in any locked patchers, so there is no need to enable it, though you may find it useful for your personal projects.

12. Close the Help file for *print*
13. Paste the contents copied from the *print* Help file into your patch using the commands ⌘+v (Mac) or ctrl+v (Windows)

Arguments

In your patch, notice that one of the *print* objects in the Help file contents you just copied contains the letter *x* next to the object name. This is called an *argument.* Arguments are used to set or change variables for an object and are related to the types of data and messages the object deals with. In most cases, specifying an argument replaces some default value, if the object had one, with a new one. In this case, the *x* argument will replace the default name "print" in the Max window.

14. Change the *print* argument from *x* to a single word of your choosing
15. Lock the patch and click the *message* box

16. Open the Max window to reveal that the word you chose has replaced the word "print"

Look at Figure 1.6 and try to determine what will happen when you click on the *message* box:

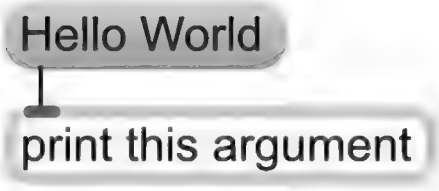


FIGURE 1.6
a print object with an argument | hello_world2.maxpat

The supplied argument is two words: *this* and *argument*. However, the *print* object only takes one argument so the second argument, *argument,* is dropped. The message in the Max window reads:

Object: this Message: Hello World

Separating Items in a Message

Now, unlock your patch and

17. Double click the *message* box containing the text *Hello World* and put a comma after the word *Hello*

The comma is used to separate items in a *message* box. The *message* box now contains two separate items. Each item in the *message* will be printed to the Max window on a separate line.

18. Lock your patch and click the same *message* box

Take a moment to save your patch as *hello_world.maxpat*. Close the patch.



FIGURE 1.7
printing a list to the Max Window | hello_world3.maxpat

Numbers: Integers and Floating Points

Create a new patcher, and

1. Create new object called *number*

The *number* object, also called a *number* box, is such a common object in Max that it has its own key command shortcut, the *i* key, for putting the object in your patch. Think of the *i* as short for *integer* as opposed to the other type of number box we'll deal with in Max: the floating-point numbers; numbers with a decimal point.

A *number* object simply holds a number.

2. Lock your patch, and click on the *number* object

Now, type the number *34* into that object. The number *34* will stay in that *number* box until you change it or close the patch. When the patch is closed, the number box will return to its default value of *0*.

3. Click on the number in the *number* box and drag the number up or down to increase or decrease the value. You may also use the arrow keys to increase or decrease the value once you have clicked in the *number* box

Unlock your patch and

4. Create 3 new *message* boxes containing the values *44*, *55*, and *42.85*, respectively
5. Connect the outlet of each *message* box to the inlet of the *number* box

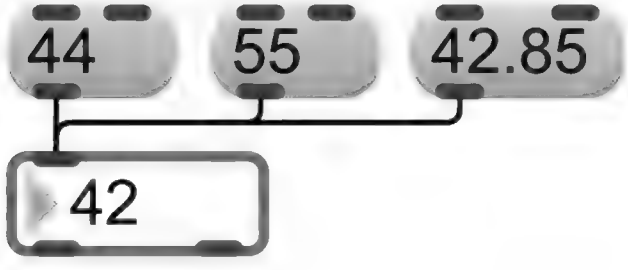


FIGURE 1.8

connecting message boxes to number boxes | number_boxes.maxpat

6. Lock your patch and click each *message* box

Notice that the value of the *number* box changes for each value you send to it by clicking a *message* box. Note that when you send the *number* box the floating point value 42.85, it drops the decimal point values completely. The *number* object only takes the integer part of a floating point number. It does not round the number up or down; it simply truncates it.

Unlock your patch and

7. Create a new object called a *flonum*

A *flonum* object is a floating-point number box. Floating-point number boxes can be used to show more precise numbers than integer numbers. Is the singer singing exactly at A 440 or is she singing at A 440.91? A *flonum* will

represent this value more accurately than a *number* box since it can contain the numbers after the decimal point unlike the *number* box. The *flonum* object, sometimes called a *float,* is such a common object in Max that it has its own key command shortcut, the *f* key, for putting the object in your patch.

8. Create 3 new *message* boxes with the values *55.72*, *44.35,* and *55,* respectively
9. Connect the outlet of each *message* box to the first inlet of the *flonum* box

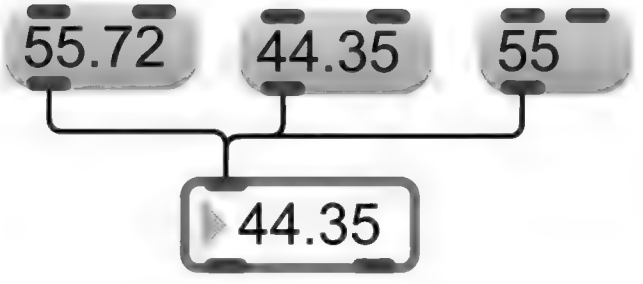


FIGURE 1.9
connecting message boxes to flonum boxes | number_boxes.maxpat

10. Lock your patch and click on each *message* box

The first two *message* boxes send the floating-point number to the *flonum* object. If you send a floating-point number with more than two decimal places, you will need to resize the *flonum* object to see these places by clicking on the button right corner of the object and dragging to the right. The third *message* box, *55*, is still received by the *flonum* even though it is not a floating-point number.

Aligning

You may have noticed that the patch cords in the above examples look more segmented than yours. This is accomplished by clicking at different points in the patch between where you initially clicked and where the patch cord connects to another object's inlet. It can also be accomplished by highlighting the patch cord by ctrl+clicking (Mac) or right clicking (Windows) it after it's connected to some objects and selecting *Align,* or by using the key command ⌘+*y* (Mac) or ctrl+shift+a (Windows).

You can also *align* highlighted objects using the same key command.[2] You can highlight multiple objects, either by holding the shift key and clicking the desired objects, or by drawing a marquee around them with the mouse. To highlight patch cords while you are drawing a marquee, hold the alt/option key.

2. This option is best used for objects positioned along the same plane. Highlighting everything in the patch and clicking *align* could make a mess.

Commenting

It is a good idea to make comments within a patch while you are writing it. As patches become larger, they can become difficult to read, and if you haven't opened the patch up in a while, it is easy to forget what you did and how certain parts of it function.

11. Create a new object called *comment*

The *comment* object, also called a *comment* box, allows you to type directly into your patch and document it. The *comment* object is such a common object in Max that it has its own key command shortcut, the *c* key, for putting the object in your patch.

12. Create 2 new objects called *comment* containing the text *integers* and *floating point numbers,* respectively
13. Label the *number* objects as integers and the *flonum* objects as floating-point numbers by moving the *comments* near these objects

In general practice, it is important to be descriptive when documenting your patch.

14. Create a third *comment* object, which we will use for the title, and enter the text *ints and floats*

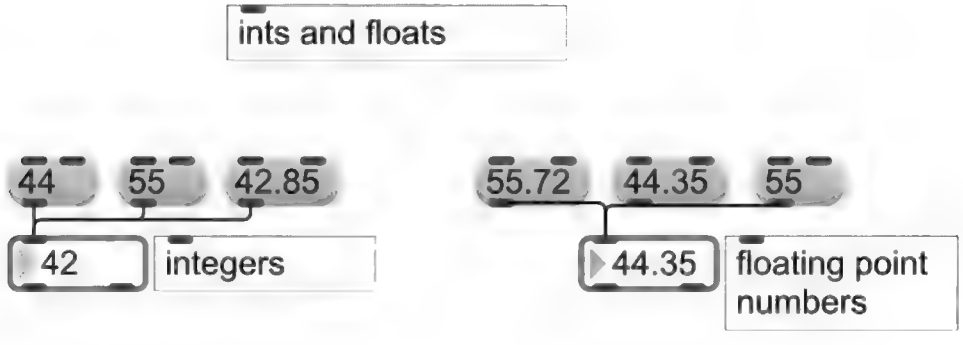


FIGURE 1.10
commenting your patch | number_boxes.maxpat

Inspector

It would be nice to see our title comment "ints and floats" in a somewhat larger font. To change properties about this or any object, go to the *Inspector* by Ctrl+clicking (Mac) or right clicking (Windows) an object and selecting *Inspector* from the contextual menu while the patch is unlocked. With an object highlighted, you can also use the key command ⌘+*i* (Mac) or ctrl+i (Windows), or click the *Inspector* icon located near the *Lock* icon at the bottom of the patch window. Another method of getting to the *Inspector* is



FIGURE 1.11
the *Inspector* icon at the center left of the object shown here in Max 5 is now an enhanced attributes wheel in Max 6.

to hold your mouse in the middle of the left side of the object and clicking the *Inspector* icon when it appears. In Max 6, this icon is an attributes wheel that provides a menu of choices of which *Object>Inspector* is one.

15. Open the *Inspector* for the *comment* box containing the title "ints and floats"

From the *Inspector* menu, you can select tabs for which to edit the object's properties. In Max 5, the tab *All* is selected by default and contains all of the property options of the other tabs. In Max 6, the *Basic* tab is shown by default, so switch to the *All* tab if necessary.

16. In the *Inspector,* scroll down to the line marked *Font Size* and increase the font to *30* by double clicking the default number and typing in the number *30*

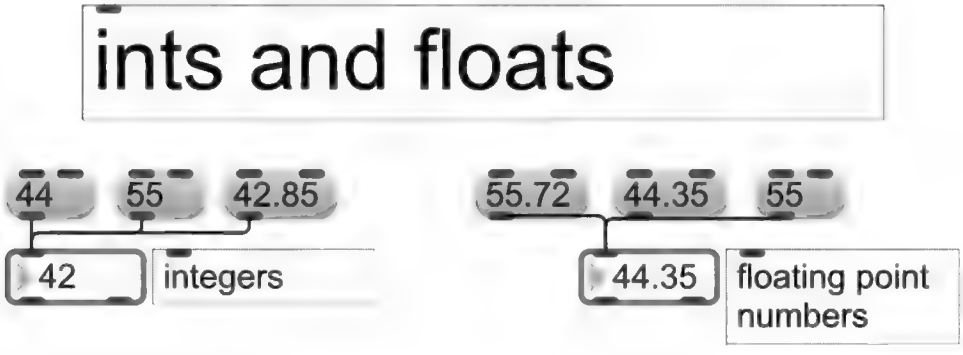


FIGURE 1.12
changing an object's properties through the Inspector | number_boxes.maxpat

Take a moment to save your patch as *number_boxes*. Close the patch.

Key Commands:

- *New Patcher*: ⌘+*n* (Mac) or *ctrl*+*n*(Windows)
- *Save Patcher*: ⌘+*s* (Mac) or ctrl+s (Windows)
- *Zoom In:* ⌘+= (Mac) or ctrl+= (Windows)
- *Zoom Out:* ⌘+- (Mac) or ctrl+- (Windows)
- *Copy*:⌘+*c* (Mac) or ctrl+c (Windows)
- *Paste*: ⌘+*v* (Mac) or ctrl+v (Windows)
- *New Object Box*: n
- *Button object*: b
- *Toggle Lock & Edit Modes*:⌘+*e* (Mac) or ctrl+e (Windows)
- *Message object*: m
- *Max window*:⌘+*m* (Mac) or ctrl+m (Windows)
- *Help*: alt/option then click an object
- *Number (integer) object*: i

- *Flonum (floating-point number) object*: f
- *Align*:⌘+*y* (Mac) or ctrl+shift+a (Windows)
- *Comment*: c
- *Inspector*: ⌘+*i* (Mac) or ctrl+e (Windows) with the object highlighted

New Objects Learned:

- *button*
- *print*
- *message*
- *number*
- *flonum*
- *comment*

Remember:

- A document in Max is referred to as a *patcher* or a *patch.*
- You can copy an object by holding the alt/option key and dragging an object.
- You can resize an object by holding your mouse over the bottom right of the object until your mouse shows two arrows at an angle, and dragging while holding the mouse button down.
- *Segmented Patch Cords* is an option that causes a patch cord to be created when you click on the outlet of an object.
- With *Segmented Patch Cords* selected, if you click on an object's outlet, the patch cord will follow your mouse until you connect it to another object's inlet or press the esc key.
- Patch cords always connect from outlets to inlets and never the other way around.
- The filename extension *".maxpat"* is given to all Max patches.
- To avoid a *Stack Overflow*, make sure objects are not feeding data back into themselves.
- The Max window shows important messages and useful debugging information in case of errors.
- Open Help by holding the alt/option key down and clicking on an object.
- To find out more information about an object, click on the *Open Reference* link within the object's Help file.
- Arguments are parameters that can be set for an object that relate somehow to the types of data that object handles.
- The *number* box truncates decimal point values completely. It does not round numbers up or down at all.

- It is important to be as descriptive as possible when commenting your patch.
- To change properties about an object, go to the *Inspector*.

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Help from the Help menu and read
  - The Basic Activities of Making Patches
  - Tips and Shortcuts
  - Tutorial Zero Video
- Select Max Tutorials from the Help menu and read
  - Hello—Creating objects and connections
  - Bang!—The bang message
  - Numbers and Lists—Types of data in Max

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Write a patch that allows a user to click on a single *button* to send 3 different *message* boxes with numerical values (integer or floating-point) to 3 separate objects: the *print* object, the *number* box, and the *flonum* box. Specify an argument for the *print* object. Comment the patch with a title in size 30 font. Use your knowledge of key commands to speed up the process.

2

Generating Music

In this chapter, we will create a program that randomly generates pitches at a specified tempo. The program will have the ability to change a number of musical variables including timbre, velocity, and tempo. We will also write a program that allows your MIDI keyboard to function as a synthesizer. These two programs will be the basis of future projects related to composition and performance.

Since you've already learned a number of objects in the previous chapter, let's agree that when you're asked to create an object that you already know, like *button*, for example, it will be sufficient for me to say "create a *button*" instead of repeating the process of creating a new object box and typing in the word *button*. Combining steps in this way will help us to get through the instructions with greater speed while reinforcing your understanding of how certain objects work. In this way, the instruction "create a message box containing the numbers 41 and 38" actually combines several smaller, and hopefully intuitive, instructions into a single step. I will slowly stop mentioning key commands and other shortcuts for objects and tasks that I've already introduced.

The RAT Patch

Create a new patch and

1. Create a new object (press *n*) called *random*

The *random* object takes a number as its only argument and randomly generates a number between *0* and one less than the argument when it receives a bang in its inlet.

2. Give this *random* object the argument *128* (Note: if you already clicked away from the object, double click it in order to, once again, enable typing within the object box)

Be sure to put a space between the word *random* and the argument *128* or else Max will look for an object called *random128* that does not exist.

3. Create a new *button* (press *b*)
4. Connect the outlet of *button* to the first inlet of *random 128*
5. Create a *number* box (press *i*)
6. Connect the outlet of *random 128* to the inlet of the *number* box

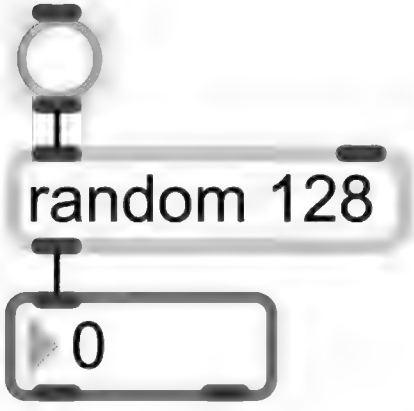


FIGURE 2.1

outputs random numbers from 0 to 127 when a bang is received | the _RAT_patch.maxpat

Do you remember that kid that you went to grammar school with who asked you to count to 10 and when you started counting "1, 2, 3 . . ." he stopped you and laughed as he declared "No! You forgot the number zero!!!" I'm sure you remember that kid, or maybe you were that kid. Well, in Max, we almost always start counting at the number 0, not the number 1. With the argument *128*, the *random* object will randomly output a total of 128 possible numbers starting at 0, which means that the range of numbers being randomly generated from *random* will be between 0 and 127.

7. Lock your patch and click the *button* to see random numbers between 0 and 127

MIDI (Musical Instrument Digital Interface)

The MIDI protocol is a language that computers use to convey musical messages. Synthesizers as well as nearly all computer music software applications deal with MIDI in some way. What are these MIDI messages?

In essence, MIDI is a bunch of messages in which a total of 128 numbers, the numbers 0–127, are used to represent musical elements like pitch and velocity. The lowest MIDI note, 0, is the pitch C at 5 octaves below middle C. The number 1 is the C♯/D♭ directly above that C, and so on. The MIDI note 12 is the C above the lowest C.

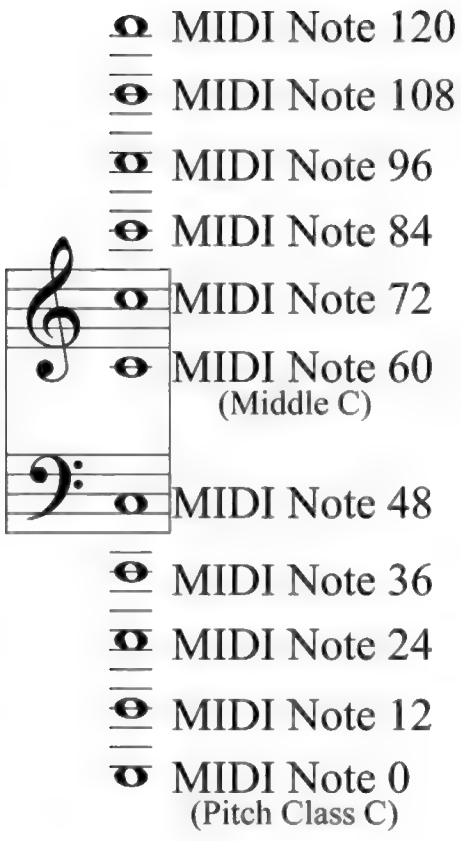


FIGURE 2.2
MIDI Note C as it relates to the grand staff

Velocity is a measure of MIDI volume represented by the same number numbers 0–127. The value 0 means that the note is off whereas the value 127 is the loudest volume that MIDI can produce. Therefore, the note 60 with a velocity of 127 could be described as middle C at a very loud volume.

Synthesizing MIDI Numbers

With the objects described earlier, we are able to generate random numbers, which is not a very musical task. However, since MIDI uses numbers to represent pitches, there's no reason we can't use those random numbers as pitches. We would then hear each random number as a random pitch. To accomplish this, we need to format those pitches into a MIDI message that includes some information about the velocity of the random pitch and how long the note will sound. An easy way to do this formatting is with the *makenote* object. The *makenote* object takes two numbers as arguments to specify a default velocity (first argument) and default duration (second argument) which will be associated with the random pitch value. The duration value is how long, in milliseconds, the note will last until *makenote* sends out a velocity *0* value to turn the note off. Unlock your patch and

8. Create a new object called *makenote* with the arguments *100* and *500*

9. Connect the first outlet of the *number* box to the first inlet of *makenote 100 500*
10. Create 2 *number* boxes
11. Connect each outlet of *makenote 100 500* to the inlet of the 2 newly created *number* boxes, respectively

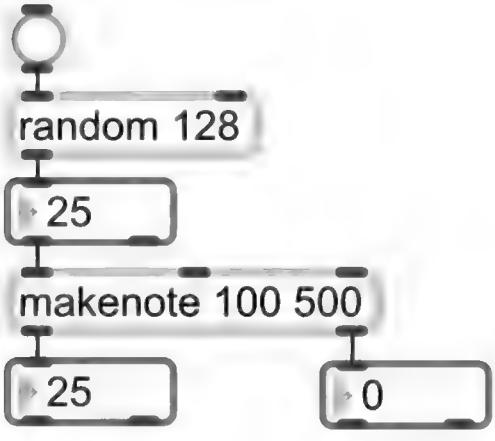


FIGURE 2.3

formats a MIDI message using the random number as pitch. | the_RAT_patch .maxpat

12. Lock your patch and click the *button*

You've likely noticed that although we see the numbers change, there is no sound. Remember, we've only now organized a bunch of messages; we haven't included an object to allow us to hear the note through our computer's soundcard. With the objects we've assembled this far, clicking on the *button* will generate a random number between *0* and *127*, the full range of MIDI pitches. The random number will be treated as the pitch in a MIDI message and partnered with a default velocity value, which we have supplied as *100*. The note will last for *500* milliseconds and then the *makenote* object will output a velocity value of *0* for that pitch turning the note off.

Look at the *number* box connected to *makenote*'s last outlet as you click the *button*. It outputs the velocity value of *100* for half a second and then sends the "note off" message: velocity 0. The *makenote* object keeps track of what numbers it receives and ensures that all notes are given a "note off"/velocity 0 message after the duration value, in this case, *500* milliseconds.

Now, let's actually hear what this sounds like by adding the object *noteout* so that *makenote* can communicate with our soundcard.

If you hold your mouse over the two outlets of the *makenote* object, you will see that it sends MIDI pitch numbers out of the left outlet and MIDI velocity numbers out of the right outlet. The *noteout* object takes pitch numbers in its leftmost inlet and velocity numbers in its middle inlet. Unlock your patch and

13. Create a new object called *noteout*
14. Connect the outlets of *makenote 100 500* to the first 2 inlets of *noteout*

The *noteout* object takes a number in its third inlet to specify the MIDI channel number. You can also specify a default argument for the MIDI channel by supplying a number in the object box after the word *noteout*. For now, we'll use the object's default MIDI channel, channel 1.

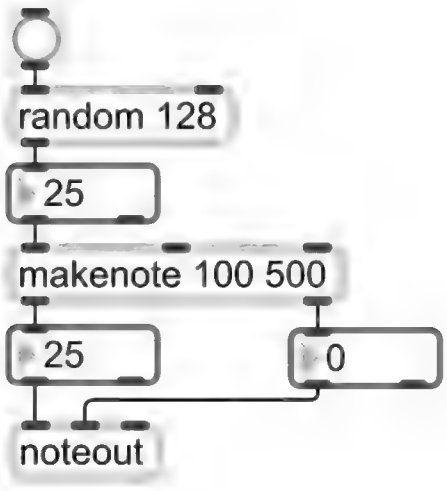


FIGURE 2.4
sends a random number out. | the_RAT_patch .maxpat

15. Lock your patch and click the *button* to generate a random number which will be synthesized as a MIDI pitch with a velocity of *100* and a duration of *500* milliseconds

Since the duration of each note lasts one half of a second, it would be nice to have the patch automatically generate a new random note each half a second.

Adding Timing

The *metro* object functions like a metronome sending out bangs at a specified interval of time. The time, in milliseconds, is specified as an argument for *metro*. In the same patch, unlock your patch and

16. Create a new object called *metro* and give it the argument *500*
17. Create a *button*
18. Connect the outlet of *metro 500* to the inlet of the *button*

When the *metro* object is "on," it will cause the *button* to blink (send bangs) every *500* milliseconds. To turn the *metro* on, we will use the *toggle* object.

The *toggle* object is an on/off switch for Max objects. Like a light switch, *toggle* has two states: on or off. The *toggle* is such a common object in Max that it has its own key command shortcut for putting the object in your patch by pressing the *t* key.

19. Create a new object called *toggle*
20. Connect the outlet of the *toggle* to the first inlet of *metro 500*

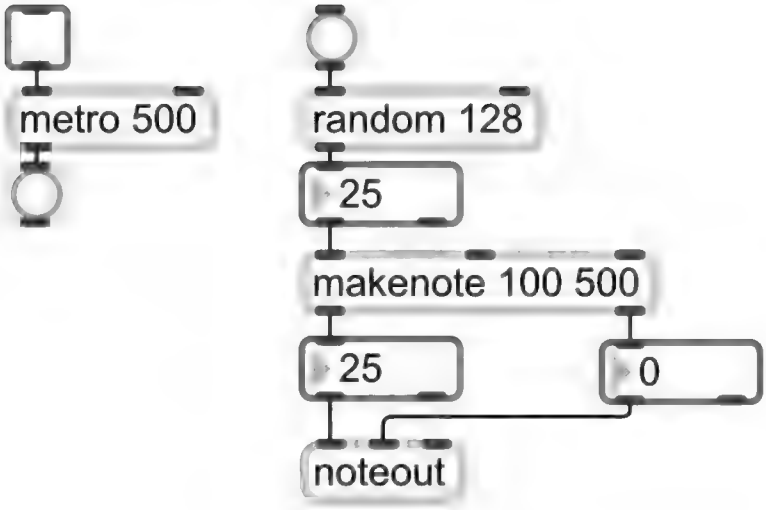


FIGURE 2.5

sends a random number out. | the_RAT_patch .maxpat

21. Lock the patch and click on the *toggle* to turn it on

An *X* means the *toggle* is on and that the *metro* object will begin sending out bang messages to the *button* every *500* milliseconds.

22. Click on the *toggle* again to turn it off

The *toggle* actually only outputs the numbers *0* and *1* where a *0* indicates that something is off and *1* indicates that something is on. We could have also sent the *metro* object a *message* box containing a *1* to turn it on, and another *message* box with a *0* in it to turn it off, but *toggle* provides a more graphical way of doing it.

Since our *metro 500* is outputting bangs, it seems we have an extra *button* in our patch. Unlock your patch and

23. Delete the *button* connected to *metro 500*'s outlet by highlighting the *button* and pressing delete
24. Connect the outlet of *metro 500* to the inlet of the *button* above the *random 128* object

With the *toggle* turned on, the *metro* will trigger a random number to become synthesized every 500 milliseconds.

25. Lock the patch and click on the *toggle* to turn it on
26. Click on the *toggle* again to turn it off

This patch creates what my students and I affectionately call *Random Atonal Trash or* the *RAT patch.*

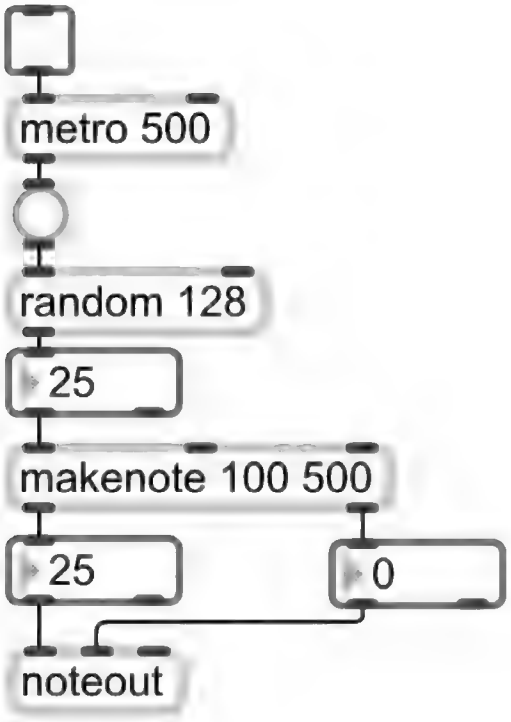


FIGURE 2.6

creates random atonal trash | the_RAT_patch .maxpat

Now that we've made our first *RAT patch*, we will discuss some of the ways that you can control this patch. Currently, the patch has only one control: the *toggle* to turn the patch on and off, which generates random pitches. However, there are many variables within the patch that could conceivably have controls. For instance, if you want to change the speed of the *metro*, you can send a number, either as a *message* box or a *number* box, to *metro*'s right inlet to replace *metro*'s default argument of *500*. Unlock your patch and

27. Create a *number* box to the upper right of *metro 500*
28. Connect the first outlet of the *number* box to *metro 500*'s second inlet

If you change the number inside the newly created *number* box by clicking in it (locked) and typing a new number,[1] the number of milliseconds *metro* will wait until it sends out another bang will change from being every *500* milliseconds to whatever number you specify. This is how you replace arguments for objects. Arguments don't change visually within objects just internally, so even though the *500* remains in the object box, *metro*'s argument will change to whatever you enter in the *number* box. Since we've "hard coded" the number *500* as *metro*'s default argument, when you open the patch, the *metro* object's default time interval will be *500* milliseconds and, thus, will send out bangs every 500 milliseconds until a new number is supplied to *metro*'s second inlet.

1. Note that the argument will actually change as soon as you click away from the object or press the return/enter key.

Since *metro*'s speed and *makenote*'s duration are both 500 milliseconds, it makes sense that if we change the argument for one, we should change it for the other or else we'd have either overlapping or staccato notes. If you hold your mouse over *makenote*'s third inlet, you will see that it receives a duration value.

29. Create a new *number* box near *makenote 100 500*
30. Connect the first outlet of this *number* box to the last inlet of *makenote 100 500*

If you specify the same argument for both the *metro* and the *makenote,* you will ensure that generated notes are legato and not staccato or overlapping.

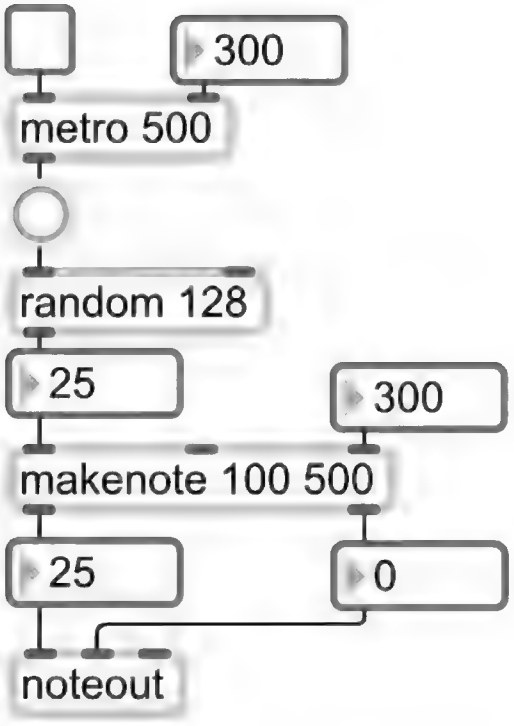


FIGURE 2.7

control your RAT patch by changing the default arguments | the_RAT _patch.maxpat

31. Lock the patch and click on the *toggle* to turn it on
32. Change the arguments for *metro* and *makenote* by clicking in the newly created *number* boxes
33. Turn the *toggle* off

This patch functions fine as it is, but what would really be useful for those who will use this patch (your users) would be some sort of graphical object that allows you to control numbers more easily than the *number* boxes alone. One such object is the *slider* object. Unlock your patch and

34. Create a new object called *slider*

When you create a new object call *slider*, the object turns into a vertical control resembling a fader on a mixing board. By default, *slider* outputs the numbers

0–127 depending on where the horizontal knob in the *slider* is positioned. If you connect the *slider* to the two number boxes that control *metro* speed and *makenote* duration, you can control both numbers simultaneously with one control.

35. Connect the outlet of *slider* to the inlet of the *number* boxes connected to the last inlet of *metro* and *makenote*

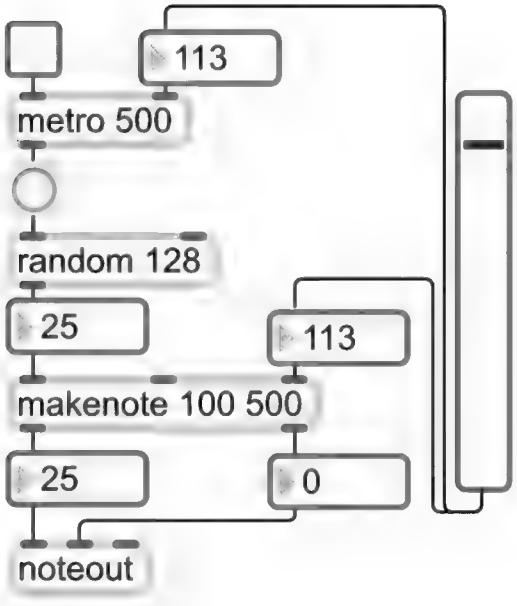


FIGURE 2.8
control your RAT patch with UI objects like slider | the_RAT_patch.maxpat

36. Lock the patch on click on the *toggle* to turn it on
37. Click on the *slider* and increase or decrease the horizontal knob to send the numbers *0–127* to the argument inlets of *metro* and *makenote*
38. Turn the *toggle* off

As I mentioned, by default, *slider* outputs numbers from 0 to 127. To increase or decrease the range of numbers *slider* outputs, open up *slider*'s *Inspector* menu. Note that the "output minimum" will offset the starting number. For example, if your *slider* range is *200* and your minimum number is 50, the *slider* will output the numbers 50–249, a range of 200 numbers. Unlock your patch and

39. Open the Inspector for *slider* and change the range of this *slider* to *1000* and leave its output minimum at the default value 0

You may feel that the notes come out too fast when the *slider* value is set too low. In fact, on some slower computers, if the *metro* object tries to send out bangs at a rate faster than around 20 milliseconds, it isn't always stable. There is a way to help the latter with something called *Overdrive* which we'll discuss in Chapter 10. To address the former concern, we can specify a minimum value for *slider* to output.

40. In the Inspector for *slider*, set the output minimum to 20

Because we want to make our patches foolproof, it's a good idea to put in default arguments and limits to restrict the range of numbers to known or desired values. To do this,

41. Open the *Inspector* for the *number* box connected to *metro*'s last inlet and set the *minimum* value to 20
42. Do the same as the previous step for the *number* box connected to *makenote*'s last inlet

Save this patch as a "the_RAT_patch.maxpat." Before you close the RAT patch, highlight and copy the *makenote* and *noteout* objects, as we'll paste them into a new patch. This brings up a good technique: reuse working parts of patches whenever possible.

Algorithmic Composition

The term *algorithmic composition* is used to describe music that is composed using some sort of process. It commonly refers to music composed with the use of computer algorithms or processes. However, algorithmic composition is as old as composition itself.

Guido D'Arezzo, for example, notated a scale on the staff and wrote a different vowel beneath each pitch. Then, using some Latin text, he matched the vowel sound of each syllable in the text to the pitch in his scale that used the similar sounding vowel. Imagine conducting a similar compositional activity using the patch you just made as a tool where, instead of using the vowels *a, e, i, o,* and *u*, you used the numbers 1 through 8.

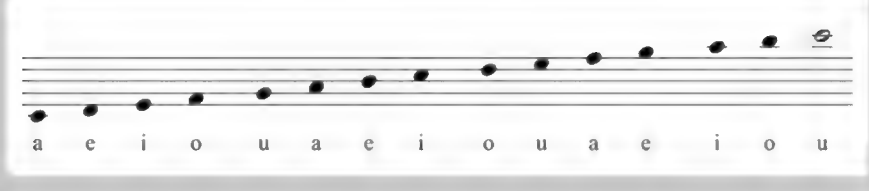


FIGURE A
Guido D'Arezzo-style algorithmic composition

W. A. Mozart's *Musikalisches Würfelspiel* was another type of algorithmic technique used to compose a minuet by reordering pre-composed measures of music. This 18th-century technique of recombinance is similar to the *Experiments in Musical Intelligence (EMI)* described by David Cope (1991; 1996; 2000) in which new compositions were created in the style of composers such as J. S. Bach, Chopin, and Beethoven through an algorithmic process data-driven by the works of these composers.

In *Interactive Music Systems*, Robert Rowe (1993) describes 3 methods of algorithmic composition:

Generative methods use sets of rules to produce complete musical output from the stored fundamental material.

Sequenced techniques use prerecorded music fragments in response to some real-time input. Some aspects of these fragments may be varied in performance, such as tempo playback, dynamic shape, slight rhythmic variations, etc.

Transformative methods take some existing musical material and apply transformation to it to produce variants. According to the technique, these variants may or may not be recognizable related to the original. For transformative algorithms, the source material is complete musical input.

All compositions are generally in some nature algorithmic even if they do not use a computer; a computer, however, can help to make the composition process much more efficient. Estonian composer Arvo Pärt's *Cantus in Memoriam Benjamin Britten* (1977) is a prolation canon in which each instrument plays the descending notes of the *a minor* scale at different harmonic speeds. In this composition, once the composer decided on this process, the nature of composing the piece on staff paper was largely a matter of adhering to the process and putting the notes into the right place. Of course, there are numerous musical judgments that the composer also made in addition to carrying out the process. The point of this illustration is that when the composer decided on a process, he had at least two choices: do it by hand, which he did, or, if he had known a programming language, use a computer to carry out the task. The latter would have been much more efficient and would have yielded the same piece.

Though you may not be interested in composing in the style of Pärt or Cope, we can certainly glean from the idea that technology can assist with composition in ways that can make doing traditional compositional processes much more efficient. Technology, and Max in particular, can also be used to develop compositional processes that are both anchored in tradition and radically new.

According to one study by Strand (2006) detailing the composition methods used in the classroom by music educators, 5.9% of the 339 educators surveyed reported using composition often in the classroom, 39.8% reported using composition "sometimes," 19.5% reported using composition "rarely," 23% reported using composition "very rarely," and 11.5% reported never using composition. The most common reason given for not using composition was that there were too many other learning activities to include composition in the classroom (56.9%). The lack of access to technology was the second most common reason (28.2%). Strand's research suggests that composition may be under-taught in classrooms.

In future chapters, we will continue to look at the ways that Max can be used to allow individuals to compose and perform music, as well as the many ways that Max can help educators to teach composition and theory.

Slider Patch

Close the old patch and create a new patch.

1. Paste *makenote 100 500* and *noteout* into the new patch or simply create these 2 objects
2. Connect the 2 outlets of *makenote 100 500* to the first 2 inlets of *noteout*
3. Create a new object called *slider*
4. Connect the outlet of *slider* directly to *makenote 100 500*'s left inlet



FIGURE 2.9
parts of the RAT patch reused | the_RAT_reused .maxpat

5. Lock the patch and increase or decrease the *slider* position

In this patch, the numbers 0–127 flow directly from *slider* to *makenote* to *noteout* without the need for any *number* boxes. Though this may save a few seconds in the creation of a new patch, it is good practice to use *number* boxes after

every user interface (UI) object and other objects where the actual numbers you are working with are unclear. If something in your patch is not working right, it's much easier to locate the problem if you can see where numbers are, or are not, going. Save the patch as *the_RAT_reused.maxpat* and close the patch

Rat Patch 2

Reopen the *RAT patch* you created earlier or, for good practice, rebuild the *RAT patch* from memory and select *Save As* from the *File* menu at the top. Name the file *the_RAT_patch_2.maxpat.*

As musicians, we are not used to thinking about musical time in terms of milliseconds; the *tempo* object would be more appropriate as it allows us to discuss time in beats per minute, multiples of the beat, and division of the beat into notes (quarters, eighths, sixteenths, etc.). Unlock the patch and

1. Delete the *slider* from this patch
2. Double click the *metro 500* object and highlight and replace the *metro 500* with the word *tempo* with the arguments *120* for initial tempo in beats per minute (BPM), *1* for beat multiplier, and *8* for beat division, in this case, the eighth note.

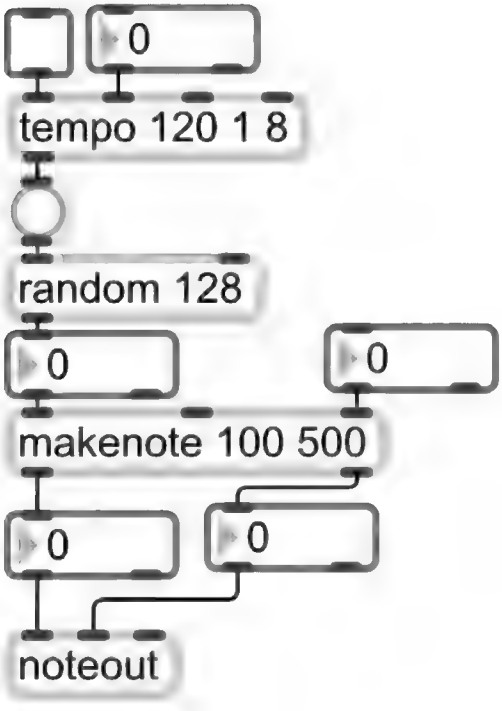


FIGURE 2.10

the RAT patch controlled in BPM instead of milliseconds | the_RAT _patch_2.maxpat

Our new patch with the *tempo* object functions similarly to the original RAT patch, but we now have the ability to work with timing in terms of beat values, values that musicians are typically more accustomed to working with.

Currently, the *tempo* will send a bang on every eighth note at 120 beats per minute. The number box that formerly controlled the *metro* time interval can now be used to control the BPM value for *tempo*. We know, from our discussion of changing default arguments, that the *8* indicating eighth notes can be changed, so let's give our patch the ability to beat whole, half, quarter, eighth and sixteenth notes.

3. Create 5 *message* boxes (press *m*) containing the number *1* for whole notes, *2* for half notes, *4* for quarter notes, *8* for eighth notes, and *16* for sixteenth notes, respectively
4. Connect the outlet of each of these 5 *message* boxes to the fourth inlet of the *tempo* object (the inlet that corresponds to the beat division value)

Other beat division values are possible, but let's start with these numbers.

5. Lock the patch and turn the *tempo* object on via the *toggle*
6. Change the beat division of *tempo* by clicking the different *message* boxes you created
7. Turn off the *toggle*

Although we know what will happen to the tempo when you click the message *8*, the average user of your patch may not understand how to use the control. Let's take a moment to make some more articulate controls to our patch. Unlock your patch and

8. Create 5 new *message* boxes containing the text *Whole Note, Half Note, Quarter Note,* and *8th Note,* respectively

The *tempo* object will not understand these much more descriptive *message* boxes, so we will connect these *message* boxes to the first inlet of the numerical *message* boxes you created earlier. When we do so, clicking on the descriptive *message* box will send a bang to the numerical *message* box that *tempo* can interpret.

9. Connect the outlet of the *message* box containing the text *Whole Note* to the first inlet of the *message* box containing the number *1*
10. Connect the outlet of the *message* box containing the text *Half Note* to the first inlet of the *message* box containing the number *2*
11. Connect the outlet of the *message* box containing the text *Quarter Note* to the first inlet of the *message* box containing the number *4*

12. Connect the outlet of the *message* box containing the text *Eighth Note* to the first inlet of the *message* box containing the number *8*

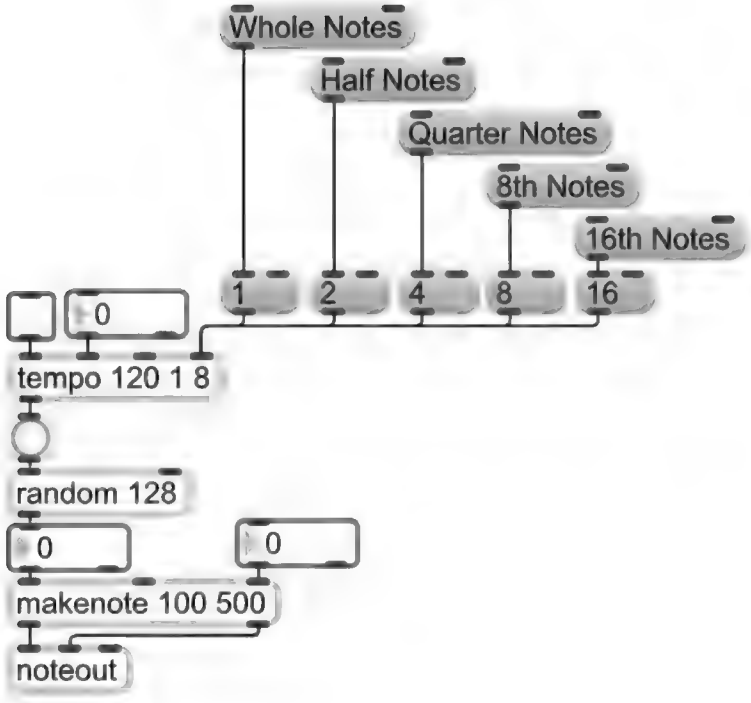


FIGURE 2.11
message boxes trigger beat divisions of the tempo object | the_RAT _patch_2.maxpat

13. Lock the patch and turn the *toggle* on
14. Change the beat division of *tempo* by clicking the descriptive *message* boxes you created
15. Turn off the *toggle*

We can create an even more descriptive *message* box that triggers both a tempo change and a beat division change by connecting its outlet to other *message* boxes containing the desired arguments. Unlock your patch and

16. Create a *message* box containing the text *8th Notes at Largo*
17. Create 2 *message* boxes containing the text *50* and *8*, respectively
18. Connect the outlet of the *message 8th Notes at Largo* to the first inlet of the *message* boxes *50* and *8*, respectively
19. Connect the outlet of *message 50* to the *number* box connected to *tempo*'s second inlet
20. Connect the outlet of *message 8* to *tempo*'s last inlet

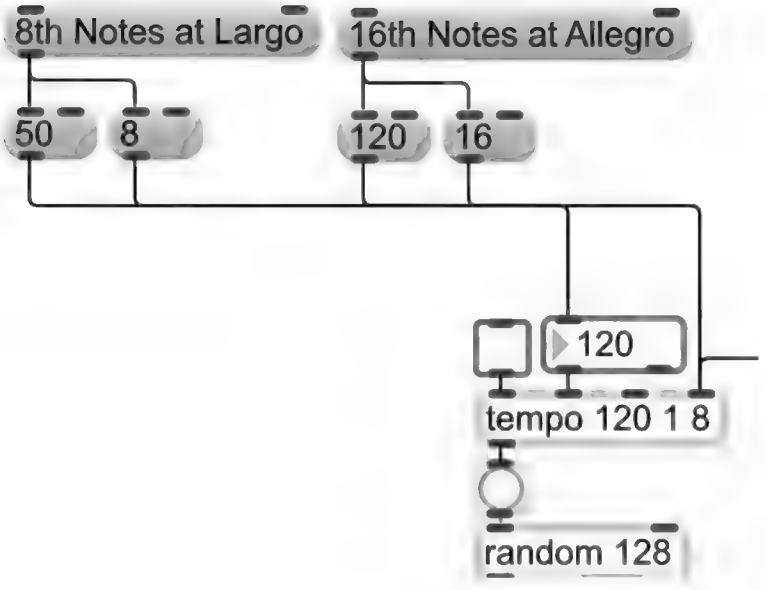


FIGURE 2.12

message boxes trigger multiple argument changes to tempo | the_RAT_patch_2.maxpat

1. Lock the patch and turn the *toggle* on
2. Click on the *message 8th Notes at Largo* to trigger the value *50* to be sent as an argument for the tempo and *8* to be sent as an argument for the beat division
3. Turn off the *toggle*

Note that instead of a descriptive *message* box, a single *button* could also have been used to trigger the change of multiple values. Save and close this patch.

MIDI Input

The next section involves getting MIDI data into Max using a MIDI controller of some kind. If you have a MIDI keyboard or other device, take this time to connect it to your computer. You will need to close and restart Max if your device was not connected or powered up when you opened Max.

We know how to get MIDI data out of Max using the *noteout* object. We can get MIDI data into Max using the *notein* object. Create a new patch and

1. Create a new object called *notein*
2. Create 3 *number* boxes
3. Connect each of the 3 outlets of *notein* to the inlet of each *number* box, respectively
4. Create a new object called *noteout*
5. Connect the first outlet of each of the 3 *number* boxes to the 3 inlets of *noteout,* respectively

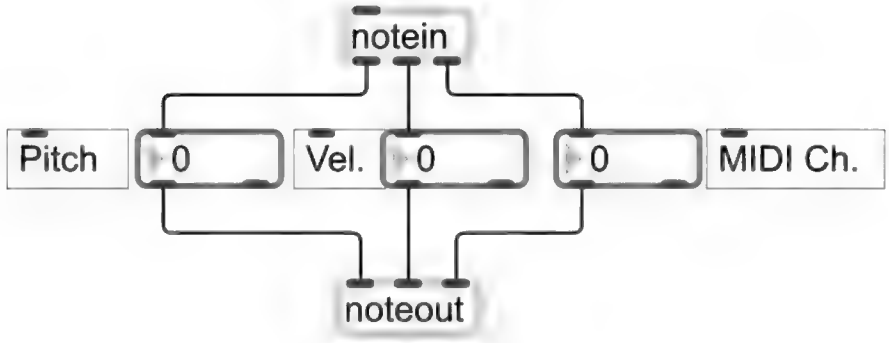


FIGURE 2.13
basic MIDI device input | MIDI_IO.maxpat

As you play notes on your MIDI device, you can see the numerical MIDI values for pitch, velocity, and MIDI channel displayed in the 3 *number* boxes. On your MIDI device, play middle C at a soft volume. The first *number* box will read *60* and the second *number* box will read some value between 0 and 127 depending on how softly you played the note. Notice that when you take your hand off the keyboard, you send out a velocity value of zero for whatever pitch(es) you were playing. These velocity zero messages are commonly referred to as "note off" messages. Even if you play just one key on your keyboard, there are multiple numbers that are expressed as part of the MIDI message for that note.

The *notein* and *noteout* objects communicate with your computer's soundcard and any available MIDI devices. When the patch is locked, you can double click on one of these objects to see a list of all MIDI devices currently connected to your computer.

MIDI data are sent on streams called *channels*. This allows a user to send musical data like pitches and velocity for separate instruments through a single cable without all of the separate MIDI data being merged into one instrument. Data on each MIDI channel can then be assigned to use different timbres and can even be routed to other synthesizers.

In MIDI, a *program* refers to the type of timbre used in the synthesis of MIDI pitches and velocities. Before synthesizer manufacturers agreed on certain standards, later known as General MIDI, there were inconsistencies among the ways these manufacturers organized their MIDI programs. If your favorite piano sound was located at program number 001 on one keyboard, that same program number on another keyboard might be a trumpet sound.

When the General MIDI (GM) specification was created, the 128 MIDI programs were standardized so that synthesizer manufactures had the same timbres in the same locations. For example, GM program 0 is always "acoustic grand piano" while program 127 is always "gunshot."[2]

2. According to the GM specification, MIDI data on channel 10 will always be percussion instruments regardless of the program you select.

To change the MIDI program value in Max, use the *pgmout* object. This object allows you to send the numbers 0–127 to change the MIDI timbre.

6. Create a new object called *slider*
7. Create a *number* box
8. Connect the outlet of *slider* to the inlet of the *number* box
9. Create a new object called *pgmout*
10. Connect the first outlet of the *number* box to the first inlet of the *pgmout* object

The *pgmout* object does not need to be connected to any further objects. Like *noteout*, it communicates with your computer's MIDI output devices.

11. Lock the patch and increase/decrease the *slider* to change the MIDI program while playing notes on your MIDI keyboard

As you know, the default *slider* range is 0–127 which works perfectly with the range of numbers that *pgmout* can receive. You have now programmed a fully functional MIDI keyboard synthesizer capable of changing timbre.

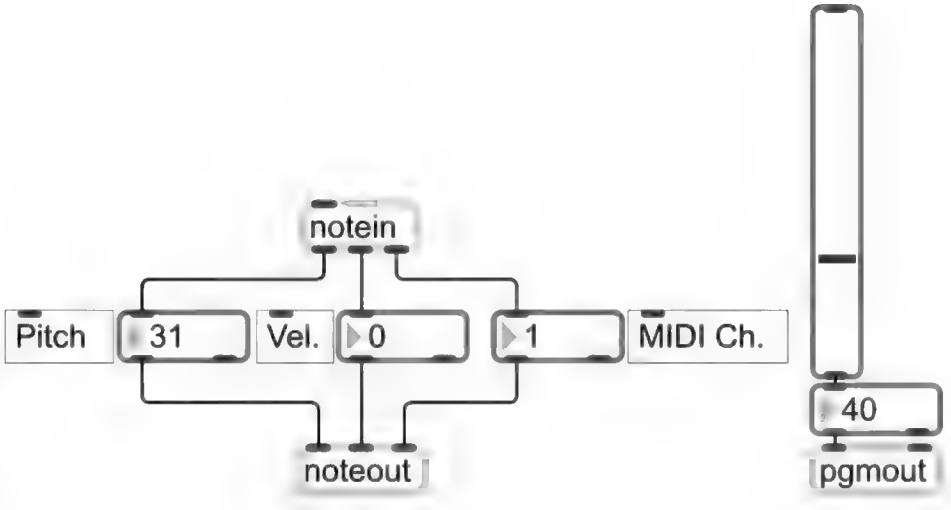


FIGURE 2.14
send MIDI program changes through pmout | MIDI_IO.maxpat

Your MIDI keyboard sends other types of MIDI data besides pitch, velocity, program, and channel. Your controller likely has two wheels on the left-hand side: one marked modulation and one marked pitch bend. These wheels send data into Max using the *ctlin* object and the *bendin*[3] object.

Let's make an instrument in which pitch is controlled by the modulation wheel and velocity is controlled by the pitch bend wheel. Unlock your patch and

3. Most MIDI controllers have one wheel that is adjusted to return to its middle state and another that is freely movable. As a result, the *bendin* object will consistently return to the same place to report a value of *64*, the middle of *128*, when the wheel is at rest.

12. Create a new object called *ctlin*
13. Create a *number* box
14. Connect the first outlet of *ctlin* to the inlet of the *number* box
15. Create a new object called *bendin*
16. Create a *number* box
17. Connect the first outlet of *bendin* to the inlet of the *number* box
18. Create a *makenote* object with the arguments *100* and *300*
19. Connect the first outlet of the *number* box receiving from *ctlin* to the first inlet of *makenote 100 300*
20. Connect the first outlet of the *number* box receiving from *bendin* to the second inlet of *makenote 100 300*
21. Create a new object called *noteout*
22. Connect both outlets of *makenote 100 300* to the first two inlets of *noteout*

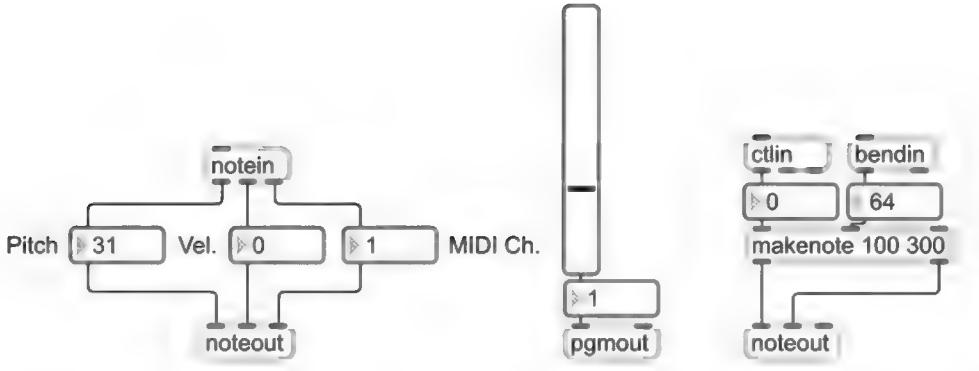


FIGURE 2.15
mapping wheel data to pitch and velocity | MIDI_IO.maxpat

You will now be able to control notes and velocity just by moving the two wheels. Give it a shot.

Since this "wheel" synthesizer is going to exist in the same patch as the functional keyboard synthesizer we made earlier, we should specify its *noteout* object to exist on MIDI channel 2 by supplying the argument *2* to *noteout*. To be consistent, let's go back and supply an argument of *1* for our keyboard synthesizer. With your patch unlocked

23. Double-click the *noteout* object receiving from the *ctlin* and *bendin* and type the number 2 next to it
24. Double-click the *noteout* object receiving from the *notein* and type the number 1 next to it

Now that our MIDI data are operating on two different channels, we can change aspects of the instrument, such as timbre, on one channel without changing the other.

25. Highlight the *slider, number* box, and *pgmout* entity and copy/paste it to the right of the *ctlin* and *bendin* entity

26. Double click the newly copied *pgmout* object on the right and type the number 2 next to it
27. Double click the *pgmout* object you made earlier (Step 9) and type the number 1 next to it

Specifying MIDI channel arguments for the *noteout* and *pgmout* objects will allow the "keyboard" synth and the "wheel" synth to use different MIDI programs simultaneously without interfering with each other because they are on separate MIDI channels, 1 and 2.

28. Lock the patch and set both *sliders* to different values, then use the MIDI keyboard and mod wheels to produce notes in some way

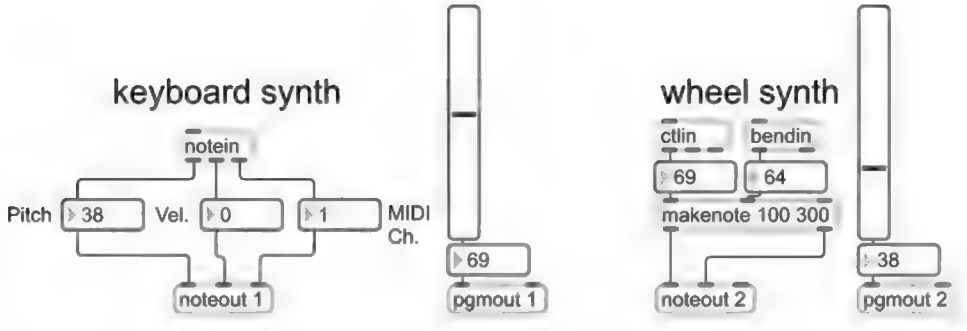


FIGURE 2.16
keyboard and wheel instruments | MIDI_IO .maxpat

In the example just shown, we were able to make a musical instrument by mapping numbers in a novel way. In traditional synthesizers, the wheels are not mapped to pitch and velocity, but since those controls are really just sending out numbers, we, as programmers, can map those numbers to anything we want.

In Max, mapping is everything. Furthermore, unlike other commercial music creation software, there are no rules in Max that dictate how a program should function. If you tried to get your favorite music notation software to interpret wheel data as pitches and velocity like we just did, the software would say "you're crazy!" In Max, it's all just numbers, and you can use those numbers in any way you want. In the next chapter, I'll be showing you ways to use those numbers to generate scales and chords.

Key Commands:

- *Toggle:* t

New Objects Learned:

- *random*
- *makenote*

- *noteout*
- *metro*
- *toggle*
- *slider*
- *tempo*
- *notein*
- *pgmout*
- *ctlin*
- *bendin*

Remember:

- User interface (UI) objects can be used to control numbers more easily than the *number* boxes alone.
- Make patches foolproof by putting in default arguments and limits to restrict the range of numbers to known or desired values.
- Reuse working parts of patches whenever possible.
- Use *number* boxes after every user interface (UI) object and other objects where the actual numbers you are working with are unclear.
- *Message* boxes can be used like *button* to trigger events in some cases.
- Assign MIDI instruments to separate channels so their data do not interface with each other.

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Metro and Toggle—Creating automatic actions
  - Basics—Getting MIDI input and output

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a RAT patch such that a user can use a *slider* to change the velocity of the random notes and a MIDI keyboard modulation wheel to control that *slider*. Set another *slider* to control the MIDI timbre. Set another *slider* to control the *tempo* from 50 BPM to 200 BPM. Be sure to document the patch thoroughly.

3 Math and Music

In this chapter, we will discuss some of the math used in musical operations. If you just cringed when I said the *m* word, have no fear. We'll be looking at the math behind musical things you already know how to do like transposing music by some interval and adding chord tones to a root note. We will also look at some of the things that will help make your program look better and more accessible to users.

By the end of this chapter, you will have made a program that harmonizes MIDI notes.

Math in Max

Let's build a MIDI synthesizer. You remember how to do this. Create a new patch and

1. Create a new object called *notein*
2. Create 2 *number* boxes
3. Connect the first 2 outlets of *notein* to both *number* boxes
4. Create a new object called *noteout*
5. Connect the first outlet of each *number* box to the first 2 inlets of *noteout*

Done! This is just about as simple as it gets for building a MIDI synthesizer. In fact, you don't even really need the number boxes; they're just displaying the pitch and velocity data as they come in. However, for now, we'll leave the *number* boxes in to ensure that data are flowing properly. Remember that if you forget which outlet connects to which inlet, you can hold your mouse over an

inlet or an outlet to reveal a small window displaying the type of data that is being received or sent.

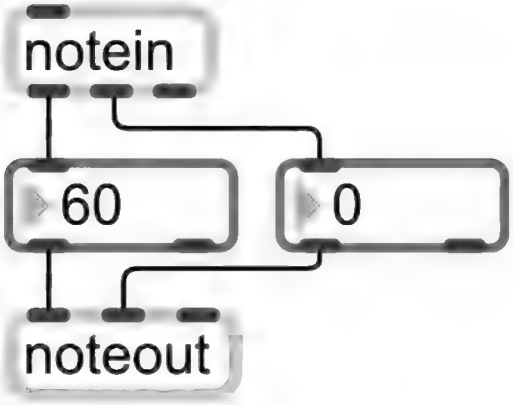


FIGURE 3.1
notein/noteout combination

6. Create a new object called *kslider*

Kslider is graphical object that resembles a keyboard. It has two inlets that receive pitch and velocity, respectively.

7. Create 2 *number* boxes beneath *kslider*'s 2 outlets
8. Connect the outlets of *kslider* to the inlet of each *number* box, respectively

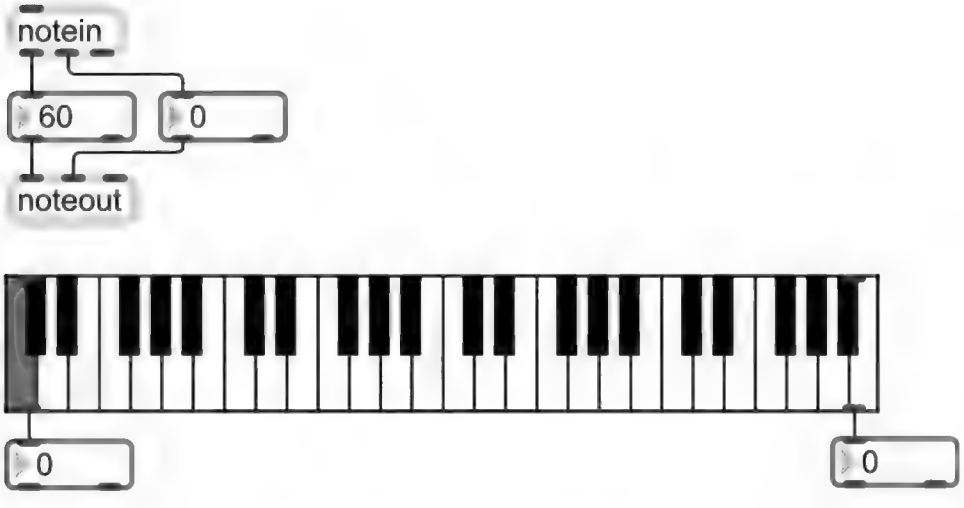


FIGURE 3.2
notein/noteout and kslider

In a moment, we will integrate the *kslider* into the existing objects in our patch. For now, let's

9. Lock the patch and click on *kslider*'s keys

Note that each key on *kslider*'s graphical keyboard sends the corresponding MIDI pitch out its left outlet. Depending on how high up on the key you clicked, the velocity sent out of the right outlet will vary from 0 to 127.

Let's incorporate *kslider* into our patch. Unlock your patch and

10. Disconnect the 2 *number* boxes connected to *noteout* by clicking on the patch cords and pressing delete
11. Connect those same 2 *number* boxes to the 2 inlets of *kslider*, respectively
12. Connect the 2 *number* boxes receiving numbers from *kslider* to the first 2 inlets of *noteout*[1]

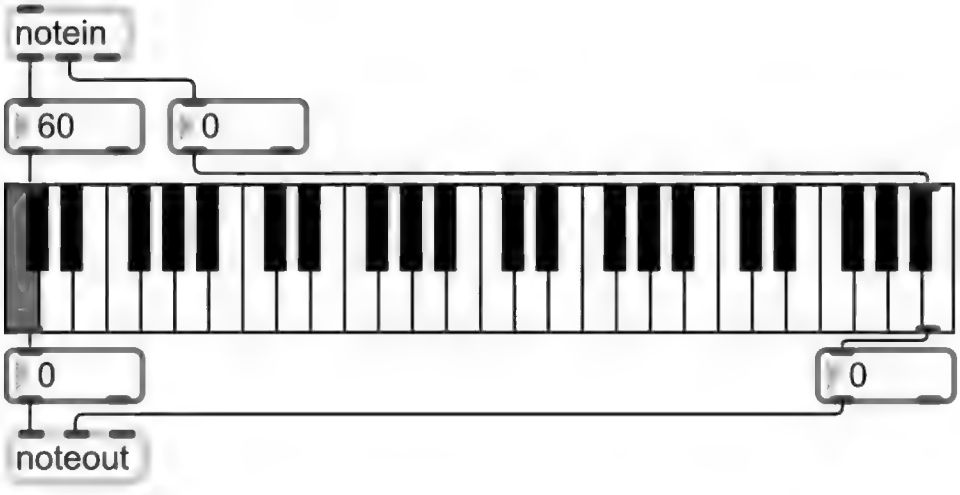


FIGURE 3.3

kslider displaying notes received through notein and passing them through to noteout

Now, as you play notes on your MIDI keyboard, they will be graphically displayed with *kslider*. Go on and play the C Major scale (if you're not a keyboardist, just play all of the white keys). You may notice, if you play two or more note simultaneously, that *kslider* only shows one note at a time. In order to change the properties of *kslider*, or any object, open the *Inspector* for *kslider* as we've done in the previous chapters.

The *Inspector* lists the option to change the display mode from monophonic to polyphonic.

13. Open the Inspector for *kslider* and switch the *Display Mode* from *monophonic* to *polyphonic* by clicking on the word *monophonic*

While you're in the inspector, it might be nice to tweak the colors a bit. If you've ever worked with little kids, you may know that sometimes the difference between a boring activity and an exciting one could hinge on whether your keyboard is of the traditional black and white variety or of the bizarre purple and green variety. For the record, I've seen grownups display this same phenomenon. A keyboard with the colors of your favorite sports team, superhero,

1. You may also move any patch cord connected to an inlet or outlet by clicking on the patch cord and dragging the colored triangle that appears at the connection to another inlet or outlet.

or cartoon character may provide some inspiration for whatever tasks you use this program for.[2]

14. Click the color swatch for the various elements of the object to change the color and then close the *Inspector*
15. Test out your patch with the great new colors by playing on your MIDI keyboard

Let's also test out the patch by sending number *messages* to *kslider*.

16. Create 2 *message* boxes and type in *60* and *100*, respectively
17. Connect the *message 60* to the first (pitch) inlet of *kslider* and the *message 100* to the second (velocity) inlet of *kslider*
18. Create a *button* above the 2 *message* boxes
19. Connect the outlet of the *button* to the left inlet of both the *60* and *100* message boxes

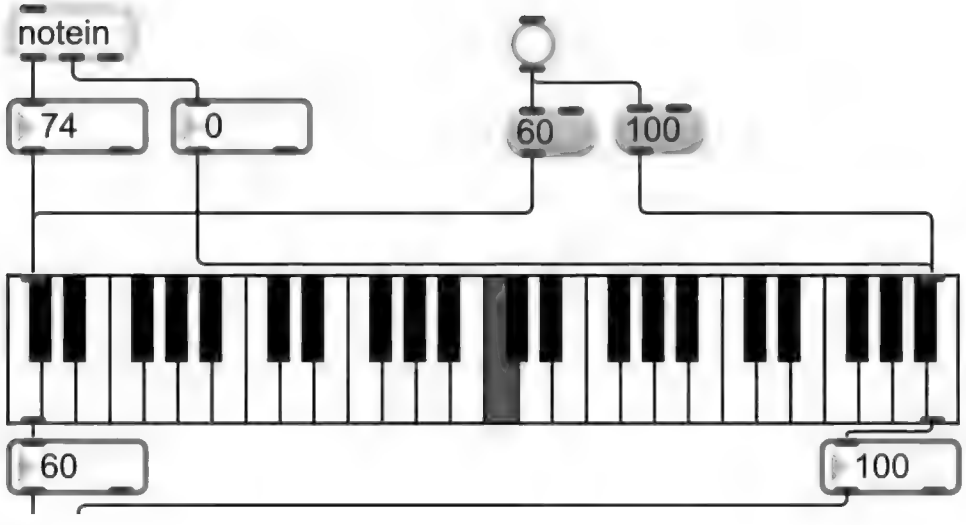


FIGURE 3.4
button triggers message boxes to simultaneously send a pitch and velocity pair to kslider

20. Lock your patch and click the *button*

When the *button* is pressed, you will hear the note middle C because we are simultaneously sending a MIDI pitch and velocity to *noteout* through *kslider*. Let's change that single note into a chord by adding more numbers to the *message* separated by a space and a comma. The commas break apart each part of the *message* so that each number is received separately, yet simultaneously, by the *kslider* producing a chord. Unlock your patch and

21. Double click on the *message* box containing *60* to change its contents to *60, 64, 67* separated by a space and a comma

2. Some users find it helpful to change the colors of related objects to identify them more easily. For instance, you may make MIDI objects like *notein* and *noteout* a certain color to distinguish them from other non-MIDI objects in the patch.

22. Highlight the *button*, and both *message* boxes (*60, 64, 67* and *100*) and copy and paste them next to the original *button* and *message* boxes. The *button* should remained connected to the *message* boxes via the patch cord after being copied
23. Double click the second *message 100* and change the number to *0*
24. Connect the outlet of both the *message 60, 64, 67* and the *message 0,* respectively, to the inlets of *kslider* as you did in Step 19

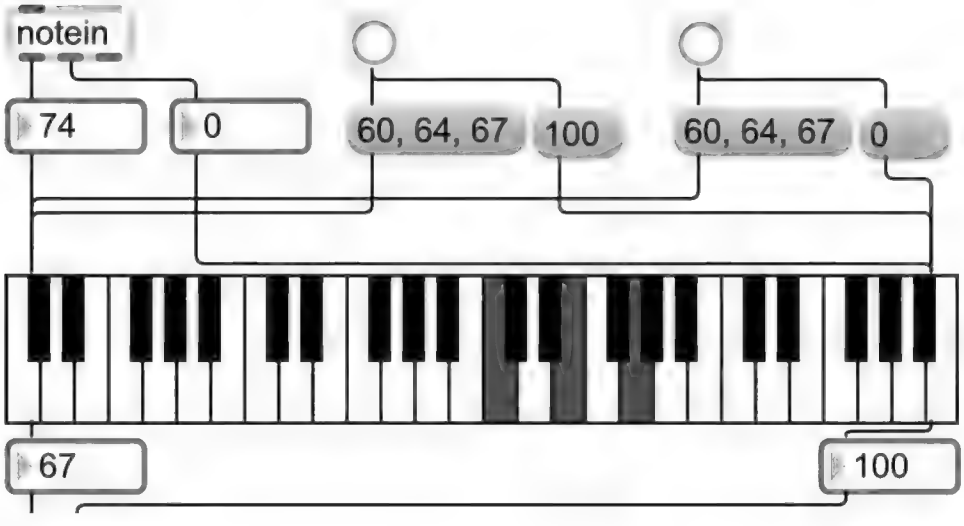


FIGURE 3.5

buttons trigger multiple pitches separated by commas and a velocity pair to kslider

You should now have two *buttons*: one triggering 3 notes with a velocity of *100* and one triggering the same 3 notes with a velocity of *0*. The second button triggering the velocity of *0* is necessary to send "note off" messages for the notes received when the first *button* is pressed. Otherwise, the notes would sustain forever; that's called a stuck note and it's very annoying. When you experience a stuck note, depending on how you wrote the program, you may need to save your work and restart Max in order to end the stuck note. In the next chapter, I will explain some types of precautions you can take to prevent stuck notes.

25. Lock your patch and click both *buttons*

You have probably noticed that the notes in the *message 60, 64, 67* are in fact the notes C, E, and G which form a C Major triad. These 3 numbers all sound simultaneously when separated with a comma and sent to the *kslider* along with a velocity message (*100*).

Forming Intervals

We're about to do some harmonization on incoming notes. When you do this, it's easy to get stuck notes, so let's take one precaution by inserting a *makenote* object into our patch. As you'll recall, a *makenote* object pairs incoming notes with a default velocity and duration. This will ensure that no matter what we do to create a stuck note, the note will last only as long as the duration value of the *makenote*.

One immediate drawback to inserting *makenote* is that note duration would now ultimately be determined by *makenote* and not by how long a note is held down on a MIDI keyboard. For example, if your *makenote* duration is *500* (milliseconds), even if you held a note down on your MIDI keyboard for longer than 500 milliseconds, a velocity *0* message (note off) would be sent for each note played 500 milliseconds after you played it. For the purpose of this chapter, we will insert the *makenote* anyway as a precaution. Unlock your patch and

26. Double click the *noteout* object and change its name to *makenote 100 500*
27. Create a new object called *noteout*
28. Connect the 2 outlets of *makenote* to the first 2 inlets of *noteout,* respectively

If you were going to explain the concept of harmony to someone, you might mention that you take a note and then add some other note above or below it. If you wanted to add a note a minor second above a note, you would add the note one semitone (half step) above the original note. To form a major second (whole step), you would add the note two semitones above the original note, and so on.

Adding notes in Max is easy with the + object. The + object takes a number as an argument. That number is then added to all numbers sent to +'s first inlet. The second inlet of + changes the value of the argument, that is, the number being added to the one sent to the first inlet.

29. Create a new object called + with the argument *0*
30. Connect the first outlet of the *number* box connected to *notein* (Step 3) to the first inlet of the + *0* object so that the *number* box is connected to both the + *0* and the *kslider*
31. Create 2 *number* boxes above and below the + *0* object
32. Connect the first outlet of one of the newly created *number* box above + *0* to + *0's* right inlet
33. Connect the outlet of + *0* to the inlet of the other newly created *number* box below + *0*

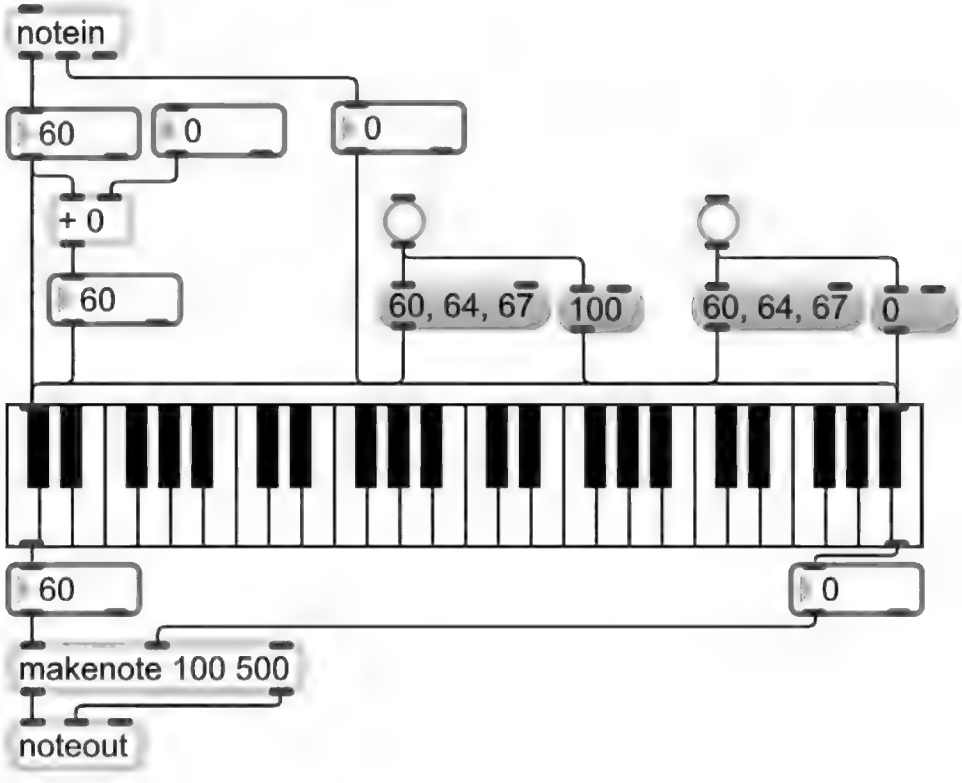


FIGURE 3.6

pitches added to the one received from notein / interval_builder.maxpat

As previously mentioned, the *number* box connected to +'s second inlet will replace the default argument which we have specified as *0*. Currently, any note played on the MIDI keyboard will be sent from *notein* to + where 0 will be added to it. This is the equivalent of adding the interval of a unison to the pitch. However, if we lock the patch and type the number 2 into the *number* box connected to + *0*'s right inlet, we will see that all notes played will have an added note that is up two semitones or one whole step from the initial pitch, the interval of a major second.

34. Lock the patch; play single notes on your keyboard
35. Change the number by which + will add notes to your initial pitches by increasing the value in the *number* box connected to + *0's* second inlet

You can now visually represent the interval of a major third by adding 4 to every note that is played. To add an octave, 12 semitones, type *12* in the *number* box. Imagine teaching a lesson on intervals where students can see the visual representation of intervals on a large colorful keyboard as they play.

If you wanted to simply transpose incoming notes without retaining the initially played note, you can simply disconnect the patch cable that connects from the *number* box directly to the *kslider*. This is similar to the "transpose" function that some synthesizers have.

Window Dressing

In 2008, I had the opportunity to study with composer and electronic music pioneer David Cope at the University of California, Santa Cruz. Among other important things I learned from him, he let me in on a little secret: "many times, programs are 1% function and 99% window dressing." The program we've been building in this chapter is functional now, but to make it really accessible to our users, we still have to do some important things to it like add controls, label the controls, document the patch, and provide instructions. All of this is not really part of the main function of the program, which is basically completed at this point, but helps to make the program user-friendly and accessible.

When I teach Max, I usually ask my students "if Grandma would be able to control the patch when she comes to visit for dinner at [insert closest holiday here]." The rationale for the humorous question is that even though she may know a bit about intervals, Grandma would most likely not understand why she has to click on a number box and type in *2* to form the interval of a major second or *4* in order to form a major third. It would be great if Grandma had some kind of menu where she could select from a list of interval types.

One such object called the *umenu* would do just the trick since it stores any kind of text in a pull-down menu. Make sure your patch is unlocked and

36. Create a new object called *umenu*
37. Open the *Inspector* for *umenu*
38. In the *Inspector,* scroll down to *Menu Items* and click the button labeled *edit* (Note: in Max 6 you must first select the *All* tab).
39. In the window that opens, type in the words *dog, cat,* and *fish,* each separated with a comma
40. Click OK and close the *Inspector*
41. Lock your patch and click on the *umenu*

Notice that all the items you entered (dog, cat, and fish) appear in the menu. Unlock your patch and

42. Create a *number* box
43. Connect the first outlet of *umenu* to the inlet of the *number* box
44. Create a *message* box
45. Connect the second outlet of *umenu* to the second inlet of the *message* box

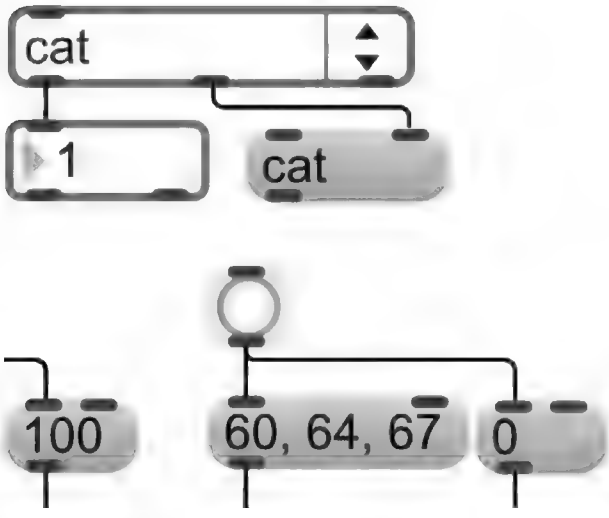


FIGURE 3.7
items listed in a umenu / interval_builder.maxpat

The *umenu* object sends the menu item number out of its first outlet and the actual text of the item out of its second inlet. The menu item number will be displayed in the *number* box you just created. The actual text of the *umenu* item will be displayed in the *message* box since any text or number sent to the second inlet of a *message* box changes the contents of a *message* box. Note that if you were to connect a *number* box to receive from the second inlet of *umenu*, the *number* box would not understand the text "dog" since it's not a number. However, if the *umenu* item was a number like *817*, the number box would understand it.

The items in *umenu* are numbered starting with *0*. There are 3 items in our menu. The first item in our menu, *dog*, is item number *0*. *Cat* is item number *1*, and *fish* is item number *2*.

As we mentioned before, what we (or Grandma) really needs in this patch is a menu with interval names to choose from. Let's now go back to the *Inspector* and change the items from *dog, cat*, and *fish*, to interval names:

46. Open the Inspector for *umenu*
47. In the *Inspector,* scroll down to *Menu Items* and click the button labeled *edit*
48. In the window that opens, remove the current items and type in the following items all separated with a comma and a space: *Unison, Minor 2nd, Major 2nd, Minor 3rd, Major 3rd, Perfect 4th, Tritone, Perfect 5th, Minor 6th, Major 6th, Minor 7th, Major 7th, Octave*
49. Click OK and then close the *Inspector*
50. Connect the first outlet of the *number* box connected to *umenu*'s first outlet to the inlet of the *number* box connected to + *0*'s second inlet

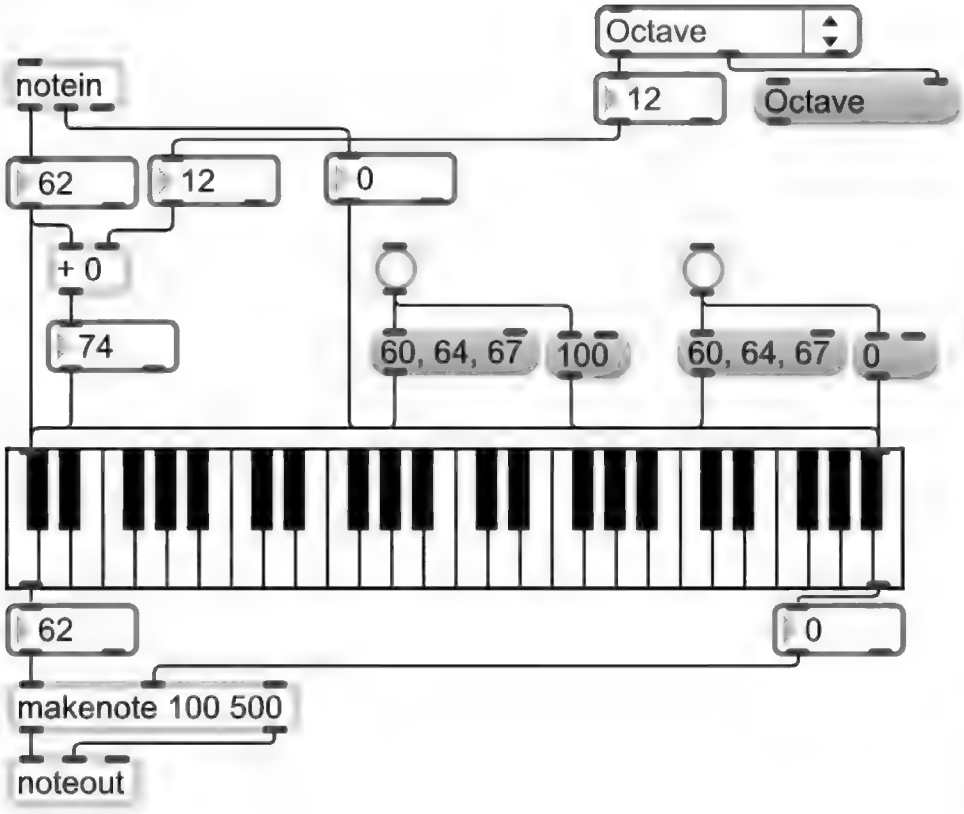


FIGURE 3.8

intervals listed in a umenu become added tone values / interval_builder .maxpat

You now have a list of every chromatic interval in one octave contained in your *umenu*. If you had liked, you could have chosen to spell major intervals with an uppercase letter "M" and minor ones with a lowercase letter "m" as is common in some theory texts. Truthfully, the spelling of these intervals is not important to *umenu*. What is important is their ordering in the *umenu*.

Item 0 is *unison*. Since we connected *umenu*'s item number so that it changes the argument by which we add intervals (second inlet of the + object), selecting *unison* from the *umenu* will add the note *0* semitones away from the initial note. Selecting *Minor 2nd* from the *umenu*, item number *1*, will add the note *1* semitone away from the initial note, which, as you may have guessed, forms the interval of a minor second.

51. Lock your patch and play a note on your MIDI keyboard
52. Select an interval name from the *umenu* and play the same note to hear the note harmonized according to the interval you selected

You will find that all of the intervals in the *umenu* will produce the menu item number that is the same as the number of semitones needed to produce the corresponding interval. Grandma will be so proud! Take a moment to save your patch as *interval_builder.maxpat* by clicking *File>Save* from the top menu.

Creating Chords

Forming triads is the next step. Think about how you explain the concept of building a major triad to a student. Perhaps you prefer to explain it intervallically: in a major chord, the third is the interval of a major third up from the root, and the fifth is perfect fifth up from the root. However you prefer to explain it, you can code it in Max by simply adding notes.

Let's start the process of building triads in Max by cleaning up our patch a bit. There are a few things left over from the demonstration of the previous patch that we really won't need in our new patch. Unlock the patch and

53. Highlight and delete the *message* boxes containing the numbers *60, 64, 67* as well as the *button*s connected to them and their associated velocity *message* boxes
54. The *number* box connected to the first outlet of *umenu* is unnecessary since it is connected to the *number* box that supplies the argument for + *0*. Highlight and delete the *number* box and connect the first outlet of *umenu* directly to the *number* box connected to + *0*'s last inlet. You should also delete the *message* box connected to *umenu*'s second outlet
55. Move the *number* box receiving velocity values from the second inlet of *notein* closer to the second inlet of *kslider* so that it is out of the way
56. Double click the *makenote 100 500* object and rename it as *noteout* (this effectively removes the *makenote* object from the patch)
57. Delete the extra *noteout* object

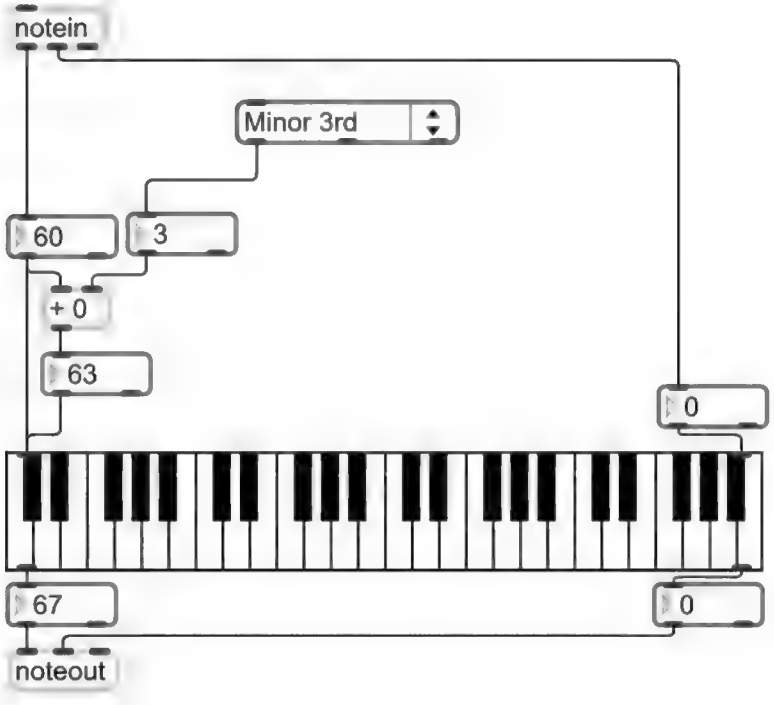


FIGURE 3.9

optimized patch / chord _builder.maxpat

Now, the patch looks much cleaner. Click *File>Save As* and save your patch as *chord_builder.maxpat*. Our patch currently has only one interval for harmonization, so let's add another interval just as we did before.

58. Create a new object called + with the argument *0*
59. Connect the first outlet of the *number* box connected to *notein* (Step 3) to the left inlet of the + *0* so that the *number* box is connected to both the + *0* and the *kslider*
60. Create two *number* boxes above and below the + *0* object
61. Connect the first outlet of one of the newly created *number* box above + *0* to + *0's* right inlet
62. Connect the outlet of + *0* to the inlet of the other newly created *number* box below + *0*
63. Copy and paste the *umenu* you created earlier and connect its first outlet to the second inlet of the newly created + *0*

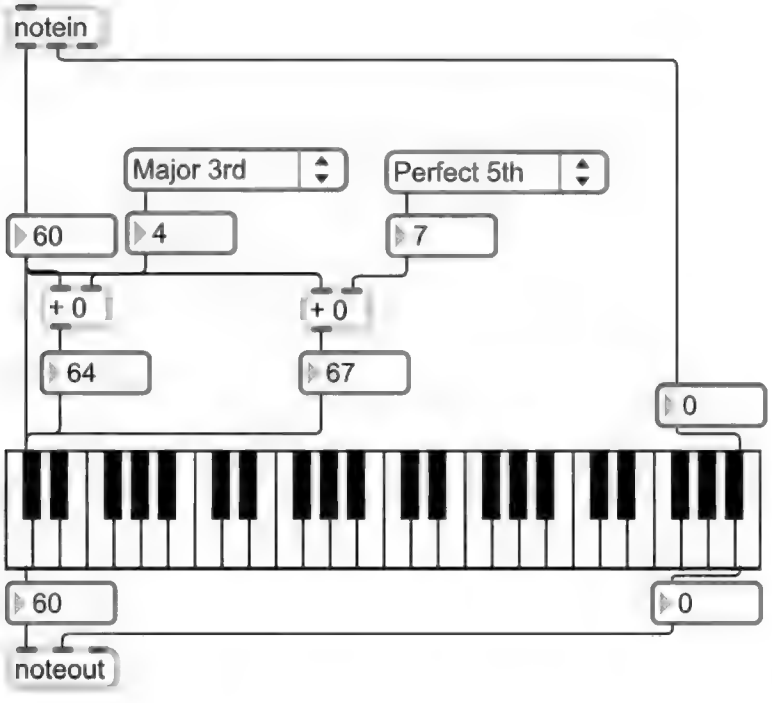


FIGURE 3.10
two intervals in umenus used to harmonize played notes / chord_builder .maxpat

You can now choose the size of both intervals you want to add to the pitch you play on your MIDI keyboard.

64. Lock your patch and select *Major 3rd* from the first *umenu* and *Perfect Fifth* from the second *umenu*
65. Play the note C on your MIDI keyboard to create a C Major Chord

This patch could be a great way to discuss the different chord qualities in a visual way. A nice addition would be a *button* to trigger different chord qualities. For

example, a user could click a *button* labeled *diminished* while play the note C on her MIDI keyboard to yield a C diminished triad. That would certainly aid in differentiating between chord qualities.

You may have noticed that *umenu* also has an inlet. If you send it a *message* box with the number *4* in it, it will immediately switch to item number *4* in the menu, *Major 3rd*. If you send a *message* box with the number *7* in it to your second *umenu*, it will immediately switch to item number *7*, *Perfect 5th*. If you were to connect a *button* to both *message* boxes, it could be used to trigger the switch of both *umenu*s simultaneously in the same way the we triggered BPM and beat division changes to the *tempo* object in Chapter 2. Unlock your patch and

66. Create 2 *message* boxes containing the numbers *3* and *4*, respectively
67. Connect these 2 *message* boxes to the first inlet of the first *umenu*
68. Create 3 more *message* boxes containing the numbers *6*, *7*, and *8*, respectively
69. Connect these 3 *message* boxes to the first inlet of the second *umenu*

You now have "shortcut" *message* boxes to retrieve the intervals from *umenu* in order to build major, minor, diminished, and augmented triads. Let's connect *button*s to these *message* boxes so that we can retrieve chord qualities by pressing a single *button*.

70. Create 4 buttons arranged horizontally
71. Connect the outlet of the first button to the first inlet of the *message* boxes containing the numbers 4 and 7 (Be sure to connect to the first inlet of the *message* box and not the second. Connecting objects to the second inlet of a *message* box changes the contained message)
72. Insert a *comment* (press the *c* key) containing the text *Major* next to this *button*
73. Connect the outlet of the second button to the *message* boxes containing the numbers 3 and 7
74. Insert a *comment* containing the text *Minor* next to this *button*
75. Connect the outlet of the third button to the *message* boxes containing the numbers 3 and 6.
76. Insert a *comment* containing the text *Diminished* next to this *button*
77. Connect the outlet of the fourth button to the *message* boxes containing the numbers 4 and 8
78. Insert a *comment* containing the text *Augmented* next to this *button*

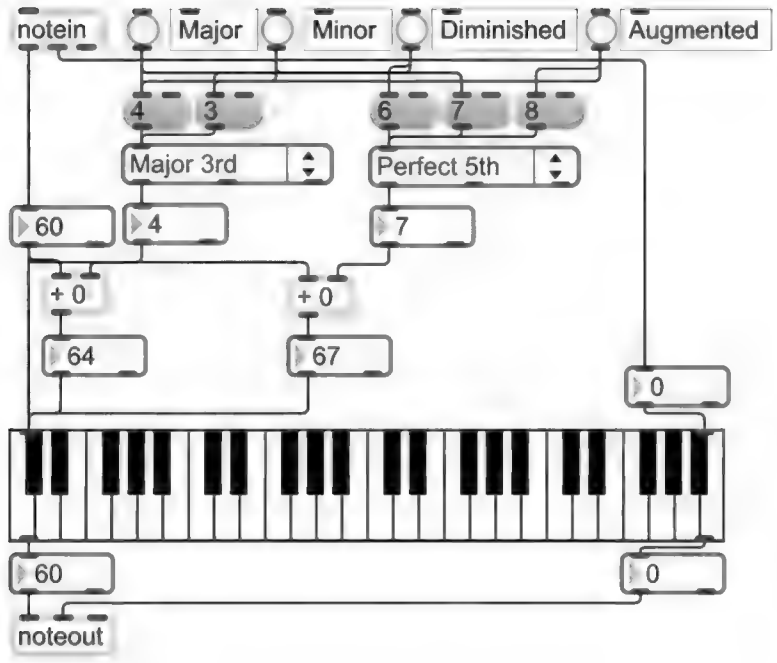


FIGURE 3.11

buttons trigger specific intervals to harmonize played notes / chord _builder.maxpat

79. Lock the patch and play a single note on your MIDI keyboard
80. Click on one of the 4 chord quality *buttons* to change the harmonization of the MIDI note

This program has now become pretty sophisticated. I suspect that a novice user could easily select a chord quality, play a root note on a MIDI keyboard, and see how the triad is built.

Presentation Mode

There are certain parts of your patch that your users don't really need to see. In fact, the most useful parts of the patch, from the user's perspective, are the *kslider*, the *umenus*, and the 4 *buttons* with their associated *comments* that trigger the changes in chord quality. In previous versions of Max, you would have had to hide less useful objects by highlighting them, selecting Object from the top menu, and selecting *Hide on Lock*. This option is still available but has been improved upon with the addition of the *Presentation Mode* or *Presentation View*, an additional layer in Max designed to be used for developing an interface for the patch. When objects are added to the Presentation mode layer, they can be moved around and reorganized without disrupting their place in the Patching mode layer.

81. Hold shift and select the following items to highlight them: *kslider*, both *umenus*, the 4 *buttons*, and their associated *comments*

82. With all items highlighted, ctrl+click (Mac) or right click (Windows) one of the highlighted objects and select "Add to Presentation." You will notice that the selected objects will then become bordered with a glowing red color

To enter Presentation mode, click the icon at the bottom menu, go to View and Presentation from the top menu, or use the key command ⌘+alt/option+*e* (Mac) or ctrl+alt+e (Windows). From within Presentation mode, rearrange the objects in a way that you feel makes for accessible use. When you are satisfied, you may return to Patching mode.

In the same way that you can use the *Inspector* to change the color and other properties for objects in a patcher, you can use the *Inspector* to change aspects about the patcher itself. To enter the *Patcher Inspector*, go to *View* and *Patcher Inspector* from the top menu or use the key command ⌘+shift+*i* (Mac) or ctrl+alt+i (Windows). You may also ctrl+click (Mac) or right click (Windows) any blank portion of a patch to bring up a contextual menu from which you can select the *Patcher Inspector*.

83. Open the *Patcher Inspector* and change the background color of your patch
84. Check the box marked next to *Open In Presentation* to allow your patch to open in Presentation mode by default when it is loaded
85. Enter a title for your patch by double clicking on the empty space next to *Title*. This title will be shown in the title portion of the program window

Further Customization

One of the best things about Max is that there are often numerous different ways to write programs that function the same. For example, if you don't like the *button*s triggering the chord change, you can use another control object. Let's explore a different way to do it with the *radiogroup* object. With your patch unlocked and in Patching mode

86. Create a new object called *radiogroup*
87. Highlight the *radiogroup* object and open the *Inspector*
88. In the *Inspector,* scroll down to number of items and change the value to *4*

The *radiogroup* object outputs a number corresponding to each of its items represented graphically by *4* circular buttons. In this case, clicking on one of the circular buttons will output one of 4 numbers, *0–3*, out of its outlet.

89. Create a *number* box beneath *radiogroup*
90. Connect the outlet of *radiogroup* to the inlet of the newly created *number* box

91. Lock your patch and click on the circular buttons in *radiogroup* to see that it outputs the numbers *0–3*
92. Put a *comment* object next to each of *radiogroup*'s *4* circular buttons labeling them *Major, Minor, Diminished*, and *Augmented,* respectively

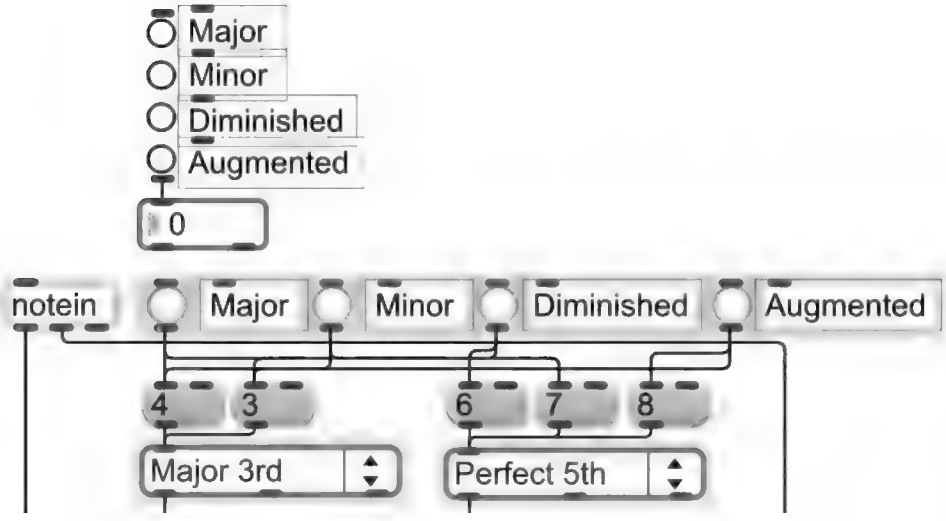


FIGURE 3.12
radiogroup sends number values / chord_builder .maxpat

To use *radiogroup* in our patch, we need some way to get *radiogroup*'s output (*0–3*) to trigger the 4 *button*s that currently control the change of different chord qualities.

One object called *select* (or just *sel,* for short) takes a number of arguments including numbers and messages. For each argument that you give to *select,* a corresponding outlet will be added to the object. If data that matches one of *select*'s arguments is sent to *select*'s left inlet, *select* will output a bang from the outlet corresponding to that argument. For example, if we give *select* the argument *1*, an outlet labeled "bang when input matches 1" will be added to the object. If we sent the number *1* to *select*'s first inlet, a bang would be sent out of *select*'s first outlet. If the input sent to select was anything other than *1*, it would be sent out of *select*'s last outlet (we will see in future chapters that this is one of the greatest features of *select*).

93. Create a new object called *select* and give *select* the following arguments: *0 1 2 3*
94. Connect the first 4 outlets of *select* respectively to the 4 *button*s that trigger the chord quality

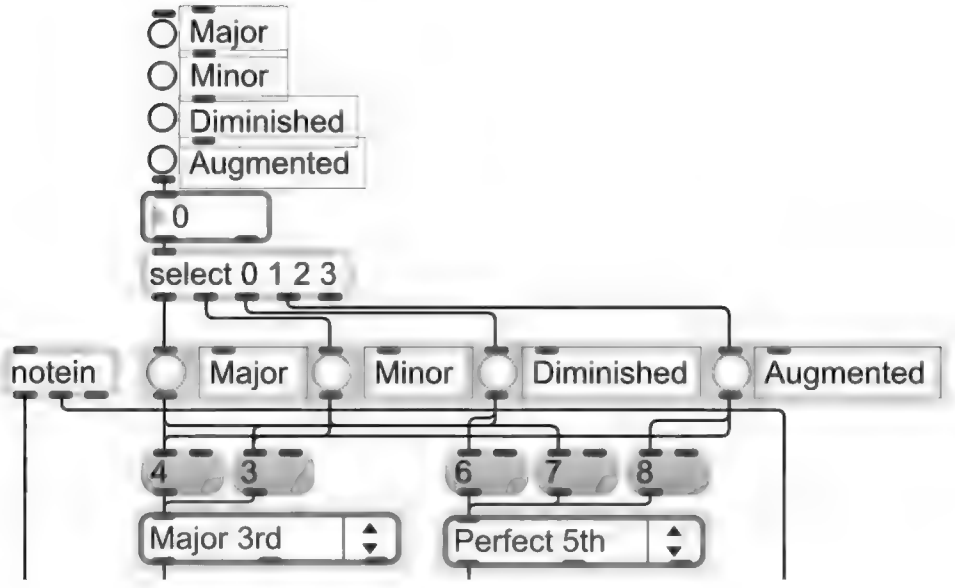


FIGURE 3.13

selecting values from radiogroup to trigger chord qualities / chord _builder.maxpat

Now, when you lock your patch and click on one of the circular buttons in *radiogroup, select* will note the item number chosen, match the number against its arguments, and send a bang to the corresponding *button* to cause the chord voicing to change.

95. Lock your patch and play a note on your MIDI keyboard
96. Select a circular button in the *radiogroup* object and play the same note

You may add the *radiogroup* and its *comments* to the Presentation mode and remove the *buttons* and their *comments* from Presentation mode if desired.

Finally, some people who use your program may desire to control chord quality in some other way besides clicking with their mouse. After all, it may be a little difficult to play notes on the keyboard with one hand and then have to move a mouse and click with the other hand. Since we already have a computer keyboard (called an ASCII keyboard) attached to the computer, we could use some of those letters, numbers, and symbols to switch between the different options in *radiogroup*. Unlock your patch and

97. Create a new object called *key*
98. Create a *number* box
99. Connect the first outlet of *key* to the inlet of the newly created *number* box

Each key on your ASCII computer keyboard is interpreted by your computer as a number even if the key you typed is a letter. The ASCII code is the numerical representation of each character on your computer keyboard. The *key* object outputs the ASCII number for the keys on your ASCII keyboard. There are a few keys that it won't allow you to use, but, in general, *key* is a great object that allows you to use your computer keyboard as a control device for your patch.

If you lock your patch and begin typing, you will see numbers fill up the *number* box beneath the *key* object. The space bar, for instance, yields the number *32*, and is a particularly purposeful key to use as a control since it's the largest one. Interestingly enough, the numbers *1–9* don't match the numbers *1–9* in ASCII code, but 49–57.

It is important to note that these numbers are sent from *key* as soon as each ASCII key is pressed. A similar object called *keyup* is like *key* but sends the numbers when the ASCII key is released.

If we want to program our patch so that the number keys *1–4* cause a change in chord quality, we can use the *select* object with the arguments *49, 50, 51*, and *52* (the ASCII values of number keys *1–4*) to detect when these keys are pressed. We can use the outputs of *select* to bang *message* boxes containing the numbers *0, 1, 2*, and *3* to the inlet of *radiogroup*. This will allow your users to change *radiogroup*'s items by clicking on its circular buttons and by using a keyboard command. Unlock your patch and

100. Create a new object called *select*. Give *select* the following arguments: *49 50 51 52*
101. Create 4 *message* boxes containing the numbers *0, 1, 2*, and *3*, respectively
102. Connect the first 4 outlets of *select* to the first inlet of each of the 4 *message* boxes, respectively
103. Connect the first outlet of each *message* box to the inlet of *radiogroup* (Note: be careful that you don't accidentally connect to the inlet of one of the *comment* objects)

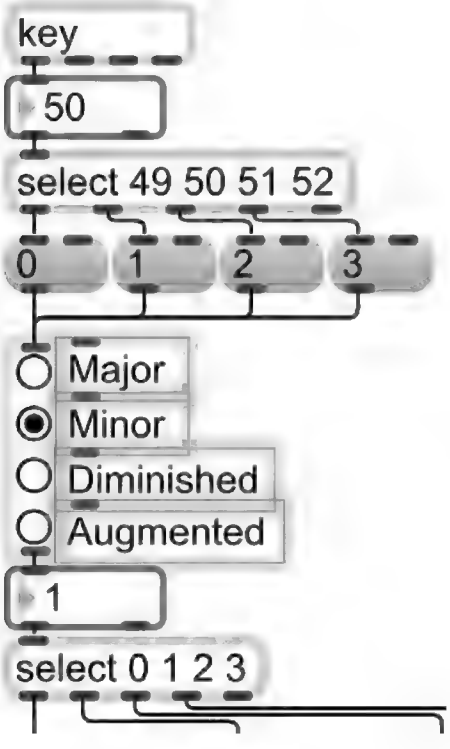


FIGURE 3.14

number keys on computer keyboard are used to trigger chord qualities / chord_builder.maxpat

104. Lock your patch and play a note on your MIDI keyboard
105. Press the number keys *1–4* on your computer keyboard to change the chord quality

Comments

Though we have primarily discussed the + object as a mathematical operation, you will find that the other math operations behave conceptually the same with respect to their specific function. The * object is used to multiply, the / object is used to divide, and the - object is used to subtract. In future chapters we will make great use of these objects and other math operators.

With the introduction of the *select* object, you now possess some of the skills to map numbers and symbols to trigger events such as the changing of chord functions. There are a few objects that function like *select* including *match* and *route*, yet each object has its own character that distinguishes it from another. In addition, objects like *key* (and *keyup*), *radiogroup, button*, *toggle*, and many other objects can become powerful controls when used with objects like *select*.

Programming in Max is a lot like *Dominoes* or the board game *Mousetrap* where very specific functions work within themselves, but, when ordered properly, cause a chain reaction of small functions that result in something great. In the next chapters, we will discuss different control options that will allow you to trigger events as well as different ways of routing that data.

Key Commands:

- Inspector: highlight an object and use the key command ⌘+*i* (Mac) or ctrl+i (Windows).
- *Toggle Presentation & Patching Modes*: ⌘+alt/option+*e* (Mac) or ctrl+alt+e (Windows)
- Patcher Inspector: ⌘+alt/option+*i* (Mac) or ctrl+alt+i (Windows)

New Objects Learned:

- *kslider*
- +
- *umenu*
- *radiogroup*
- *select* or *sel*
- *key*
- *keyup*

Remember:

- To change the properties of an object, ctrl+click (Mac) or right click (Windows) the object and select *Inspector* from the contextual menu.
- In Max, there are often numerous different ways to write programs that function the same.

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Simple Math—Performing calculations
  - Numerical User Interfaces—Sliders and dials
  - Keyboard and Mouse Input—Working with standard input interfaces

Homework:

- Read the Help file for each of the new objects introduced in this chapter. As well as the following:
  - ==
  - -
  - *
  - /
  - >
  - <
  - >=
  - <=
  - match
  - route
  - if
- Create a patch that plays and sustains a C Major triad when you press the "c" key on your ASCII keyboard and only releases the triad when you take your finger off the "c" key. Repeat the process for the F Major triad using the "f" key, and the G major triad using the "g" key. Hint: don't use *makenote* in this patch since it will not allow you to sustain notes longer than the duration you set.
- Create a patch where each letter of your first name is mapped to a chord, so that when a user types the letters on the computer's ASCII keyboard, the chords are triggered.
- Create a patch that takes ASCII keyboard numbers and uses them as pitches.

4 Scales and Chords

In this chapter, we will learn to build scales and chords and play them back in a variety of ways. Understanding how to build a scale and diatonic chords from the scale in Max will allow us to expand on these ideas in later projects resulting in the creation of new and accessible musical instruments. By the end of this chapter, you will write a program that allows users to play diatonic chords in a specified key using just the number keys on your computer keyboard.

Scale Maker

In the last chapter, you created intervals using the + object. Now, let's create a new patch that takes a note from your MIDI keyboard as a tonic, builds a major scale from it, and displays the scale on a *kslider*. Can you visualize how to build this in Max?

Let's break down the task into smaller parts: divide and conquer. In order to get data from your MIDI keyboard, you'll probably want to use *notein*.

1. Create a new object called *notein*

Next, you'll want to take the pitch coming from *notein* and add some intervals to it to form a scale. If you'll recall, a major scale is a collection of whole steps and half steps. In fact, all major scales have the same whole step and half step pattern: *whole whole half whole whole whole half*, or, in semitones, *2 2 1 2 2 2 1*. If you start on any pitch and go up a whole step to get the next note, then another whole step from that note to get the next tone, and so on, you will have built the major scale from your starting note, the tonic.

You know from last chapter that "whole step" is really two semitones which is like adding the number *2* to a note. A half step is like adding *1*. What we need for this patch is a series of + objects where whole steps and half steps are added to each note. Something like this:

2. Create a *number* box beneath the *notein* object
3. Connect the first outlet of *notein* to the inlet of the *number* box
4. Create 7 new objects called +. Give each the argument *0* and line them up horizontally
5. Create 7 *number* boxes beneath each of the + *0* objects
6. Connect the outlet of each + *0* object to the inlet of one of the newly created *number* boxes
7. Connect the first outlet of the *number* box connected to *notein* to the first inlet of the + *0* object to its right
8. Connect the first outlet of each *number* box connected to a + *0* object to the left inlet of the + *0* box to its right

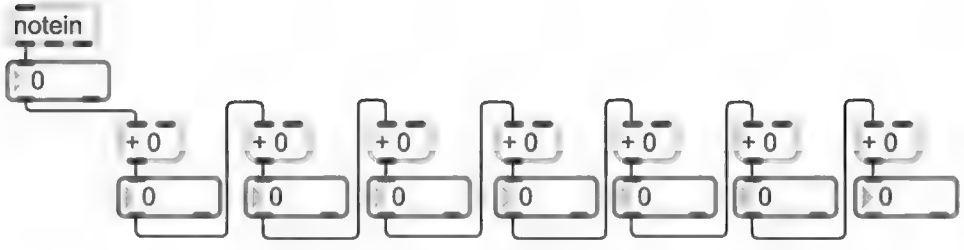


FIGURE 4.1

notein connected to + objects | scale_maker .maxpat

The way the patch is right now, when you play a note on your MIDI keyboard, the note will have 0 added to it and then be passed to the next + object where it will have 0 added to it again, and so on for each scale degree. By adding a *0* to each note, we have essentially made a scale of unisons which, as you know, is not very interesting. It's time to introduce the whole steps and half steps into the mix by changing the arguments in the + object.

Now, we can make a whole bunch of little *message* boxes containing *1* for semitone and *2* for whole tone and connect them to the second inlet of the + object, or we can try a more efficient approach by using an object called *unpack*. *Unpack* takes a list of numbers and sends each number in the list out a separate outlet. The number of arguments you give to *unpack* determines the number of outlets it will have to send data from.

9. Create a new *message* box with the following contents:
   2 2 1 2 2 1
10. Create a new object called *unpack*. Give it the argument:
    0 0 0 0 0 0 0
11. Connect the outlet of the *message* box to the first inlet of *unpack*

12. Create 7 *number* boxes beneath each of the *unpack*'s 7 outlets
13. Connect each of *unpack*'s outlets to the first inlet of one of the 7 newly created *number* boxes, respectively

When you lock your patch and click on the *message* box containing *2 2 1 2 2 2 1*, *unpack* will break up the list and send each interval to one of the 7 *number* boxes. Unlock your patch and

14. Connect the first outlet of each *number* box beneath *unpack*'s outlets to the right inlet of each + *0* object

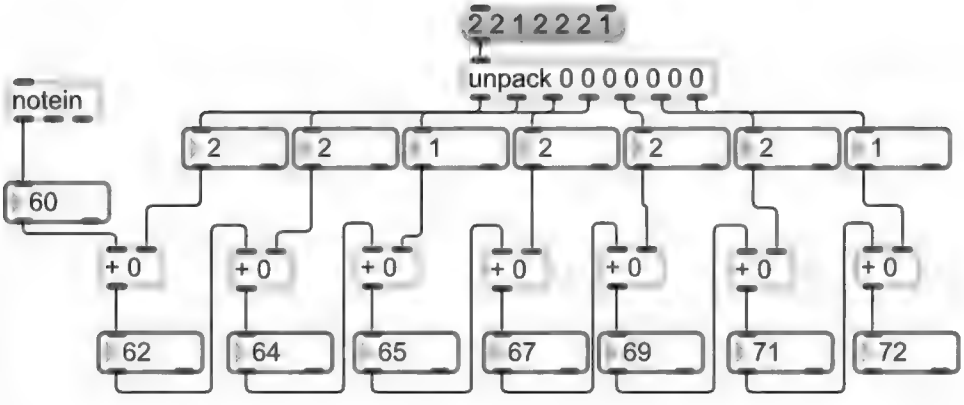


FIGURE 4.2

notein harmonized with whole steps and half steps from major scale | scale_maker.maxpat

15. Lock your patch and click on the *message* box containing 2 2 1 2 2 2 1 to *unpack* the list and replace the default argument, *0*, of each + object
16. Play a note on your MIDI keyboard

When you play a note on your MIDI keyboard, you will see the MIDI numbers for each note of a major scale using the note you played as the tonic. However, you're probably not happy with seeing notes represented numerically, so let's get them into a *kslider*.[1] Unlock your patch and

17. Create a new object called *kslider*
18. Open the *Inspector* for *kslider* and change the Display Mode to *Polyphonic*
19. In the *Inspector*, change the *Selected Key Color* to something other than the default color, then close the *Inspector*
20. Connect the first outlet of the *number* box receiving from *notein* to the first inlet of *kslider*
21. Connect the second outlet of *notein* to the last inlet of *kslider*

1. We could also open the *Inspector* for each *number* box and change the display format from *Decimal* to *MIDI* to see the numbers as pitches.

22. Connect the first outlet of the 7 *number* objects connected to + *0* (Step 6) to the left inlet of *kslider*
23. Create a *comment* box and enter the text *Tonic, Scale Degree 1* in order to label the *number* box connected to *notein*'s left outlet (Step 3)
24. Create 7 more *comment* boxes and type in *Scale Degree 2, Scale Degree 3, Scale Degree 4, Scale Degree 5, Scale Degree 6, Scale Degree 7,* and *Octave,* respectively, to label the *number* boxes connected to the + *0*'s left outlet (Step 6). Note: if you'd like to label each scale degree with the other traditional names (Supertonic, Mediant, Subdominant, etc.) feel free to do so

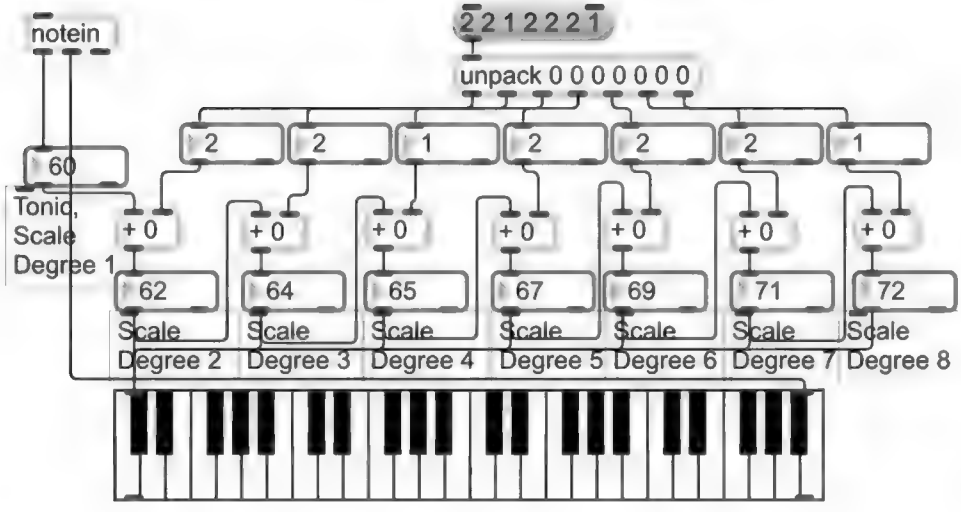


FIGURE 4.3

scale displayed in kslider | scale_maker.maxpat

Now, each time you play a note on your MIDI keyboard, you can see the notes of the major scale built from that note. If you create a second *message* box with a different set of whole steps and half steps, you will get different musical modes. For example, the whole/half step pattern for the melodic minor scale is *2 1 2 2 2 2 1*. In Chapter 6, I will show you a way to easily jump between different modes. Save this patch as *scale_maker.maxpat*, but don't close the patch.

Chord Maker

With the Scale Maker patch still open, go to *File>Save As* and save the file as *chord_maker.maxpat*. In the last chapter, we discussed building chords intervallically. Now, let's discuss building the diatonic chords from the scale.

If you asked someone to build the *1* chord in a given key, you might explain to them about our system of *tertian* harmony in which harmony is built on "thirds." Each scale degree has a chord function associated with it. The chord function is formed by adding scale degrees distanced by a diatonic third.

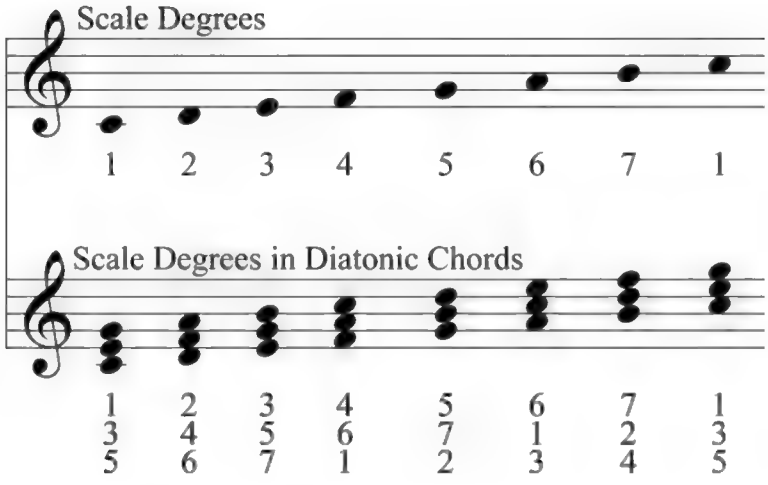


FIGURE 4.4
scales degrees as they appear in diatonic chords

Since our patch currently displays each scale degree, we can easily form the 1 chord by taking scale degree 1, the Tonic, skipping scale degree 2, taking scale degree 3 (a diatonic third from the tonic), skipping scale degree 4, and taking scale degree 5 (a diatonic third from the scale degree 3). If you wanted to build the 2 chord in a key, you would do the same process but choose scale degrees 2, 4, and 6.

Since we understand the actual musical concept, coding it in Max becomes easier: you simply need a way to work with just certain notes, scale degrees 1, 3, and 5. You might think that connecting the outlet of a single *button* to the *number* boxes containing scale degrees 1, 3, and 5 would be a good start. Clicking such a *button* would, in fact, send a bang to those 3 numbers present in the connected *number* box and send them to the *kslider*. However, it would also cause the numbers to pass through the + object which would result in all notes being sent to *kslider* as well. Since, we only want certain scale degrees to pass through and not everything, we need to think of another strategy.

An object called *int* can receive a number in its second inlet and store the integer without causing any output. The number can then be sent out of *int*'s outlet when a bang is received in its first inlet. *Int* is like a storage bin for integers.

25. Create 8 new objects called *int*. Arrange them horizontally
26. Connect the first outlet of each of the 8 *number* boxes (Step 12) to the second inlet of each of the *int* objects

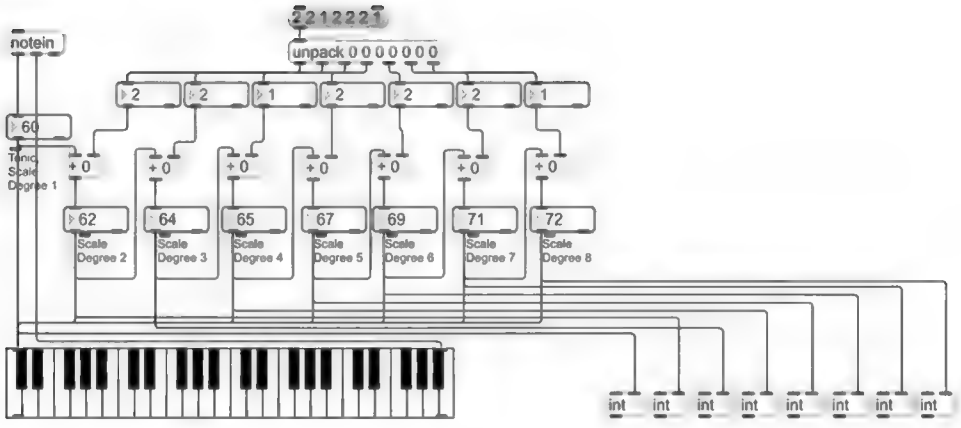


FIGURE 4.5

scale displayed in kslider | chord_maker.maxpat

Now, when you play a note on your MIDI keyboard, a scale will be built from the note according to the scale degree distance pattern of steps given to *unpack*, and the pitches of the scale will be stored in the 8 *int* objects. The next step is to cause specific *int* objects to give up their stored pitches in order to form triads. To do so, we can connect a *button* to the scale degrees we'd like.

27. Create 2 *button*s
28. Connect the outlet of 1 *button* to the first inlet of the first, third, and fifth *int* objects
29. Connect the outlet of the second *button* to the first inlet of the second, fourth, and sixth *int* objects

Clicking on one of these *buttons* will send their stored numbers out of their outlets. We won't hear anything yet because, at this point, we have not yet connected any objects to synthesize the numbers.

30. Create a new object called *makenote* with the arguments *100* and *500*
31. Connect the outlet of each *int* object to the first inlet of *makenote 100 500*
32. Copy and paste the *kslider* present in the patch beneath the *makenote 100 500*
33. Connect the 2 outlets of *makenote 100 500* to the 2 inlets of the newly created *kslider,* respectively
34. Create a new object called *noteout*
35. Connect the 2 outlets of *kslider* to the first 2 inlets of *noteout*

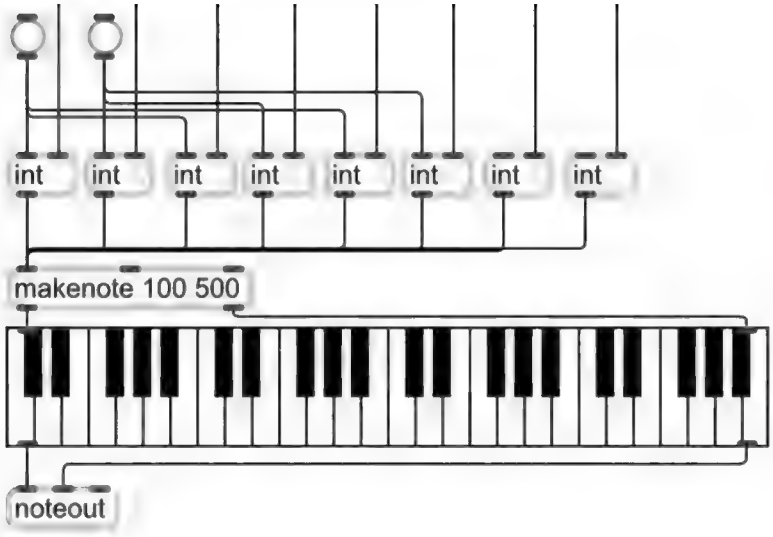


FIGURE 4.6

scale degrees stored in int objects | chord_maker .maxpat

We used a *makenote* object with a default velocity of 100 and a duration of 500 milliseconds because we are synthetically building the chord from numbers as opposed to using a MIDI keyboard. As you recall, a MIDI keyboard sends velocity 0 (noteoff) messages when the keys are lifted which is how the software knows to stop playing the notes. In our patch, we didn't program anything to do with velocity, only pitch, so we will use the *makenote* object to supply the velocity and duration values for us.

36. Lock the patch and click on 1 of the 2 *button*s to play a chord with a velocity of 100 for 500 milliseconds

At this point, you can simply connect more *button*s to the desired *int*s to create more chord options. Unlock your patch and

37. Create 6 *button*s. Arrange them horizontally to the right of the 2 *button*s you created in Step 27. There are now 8 *button*s in a row, 2 of which are currently connected to *int* objects
38. Connect the outlet of the third *button* in the row to the first inlet of the third, fifth, and 7th *int* objects
39. Connect the outlet of the fourth *button* in the row to the first inlet of the fourth, sixth, and eighth *int* objects

You probably noticed that you have run out of upper chord tones to connect *button*s to. Nothing a little math can't fix. Let's transpose some of these scale degrees up an octave.

40. Create 4 + objects with the argument *12*.
41. Connect the first outlet of the *number* boxes containing scale degrees 2, 3, 4, and 5 to the first inlet of each of the + *12* objects, respectively

42. Create 4 *int* objects
43. Connect the outlet of each of the + *12* objects to the last inlet of each *int* object, respectively

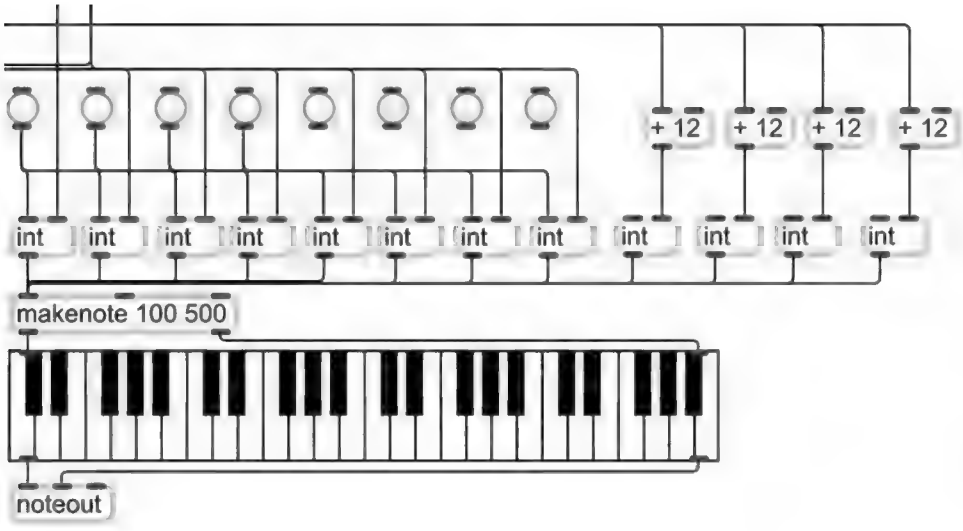


FIGURE 4.7

buttons release scale degrees stored in int objects | chord_maker .maxpat

Now that we've added duplicate scale degrees transposed up one octave to use as chord tones, you can continue connecting the remaining *button*s in thirds just as before. Be sure that you connect the *button*s to the first inlet of *int* and not to the + *12* object by accident.

44. Connect the outlet of the fifth *button* in the row to the first inlet of the fifth, seventh, and ninth *int* objects
45. Connect the outlet of the sixth *button* in the row to the first inlet of the sixth, eighth, and tenth *int* objects
46. Connect the outlet of the seventh *button* in the row to the first inlet of the seventh, ninth, and eleventh *int* objects
47. Connect the outlet of the eighth *button* in the row to the first inlet of the eighth, tenth, and twelfth *int* objects

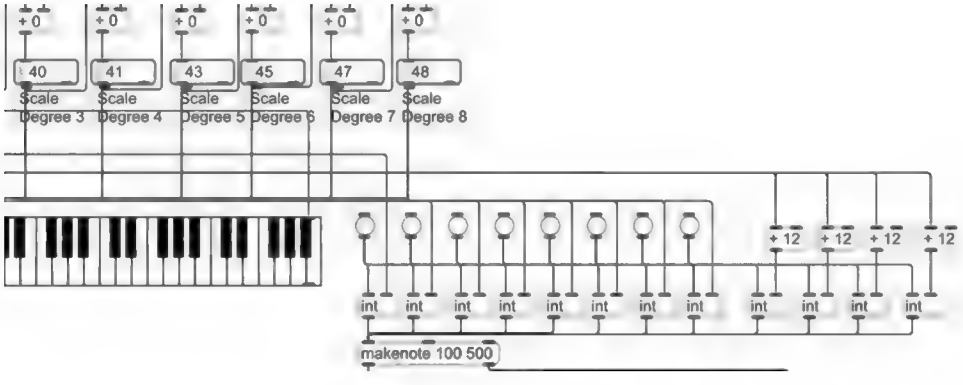


FIGURE 4.8

scale degrees are transposed 12 steps and sent to ints | chord_maker .maxpat

Each of the *buttons* will now play the diatonic chord functions.

48. Lock the patch and play a note on the MIDI keyboard to send the scale degrees to the *int* objects
49. Click on the *buttons* to trigger chord playback

If your chords do not sound correct, try to trace the patch cord from the *number* boxes to the *ints* and make sure everything is going to the appropriate inlet. You should also make sure that your *buttons* are connected to the correct *ints*. Working with so many objects and patch cords can become confusing, which is why it is extremely important to keep objects organized in a way that reflects the flow of data from one object to another, to keep objects and patch cords aligned when possible, and to *comment* different parts of the patch so that you will remember what you did and how the patch works.

Let's program this patch so that when a user presses the number *1* on her ASCII keyboard it will trigger the 1 chord in the selected key. Do you remember how we did something similar in the previous chapter? Unlock the patch and

50. Create a new object called *key*
51. Create a *number* box
52. Connect the first outlet of *key* to the inlet of the newly created *number* box

The ASCII representations of the numbers *1–8* are the numbers *49–56*. Just as in the previous chapter, we will use the ASCII numbers as the data for *select* to match in order to trigger the *8* chord *buttons*.

53. Create a new object called *select*. Give *select* the following arguments: *49 50 51 52 53 54 55 56*
54. Connect the first 8 outlets of *select* to the inlet of each of the 8 *buttons* that triggered the chord playback, respectively

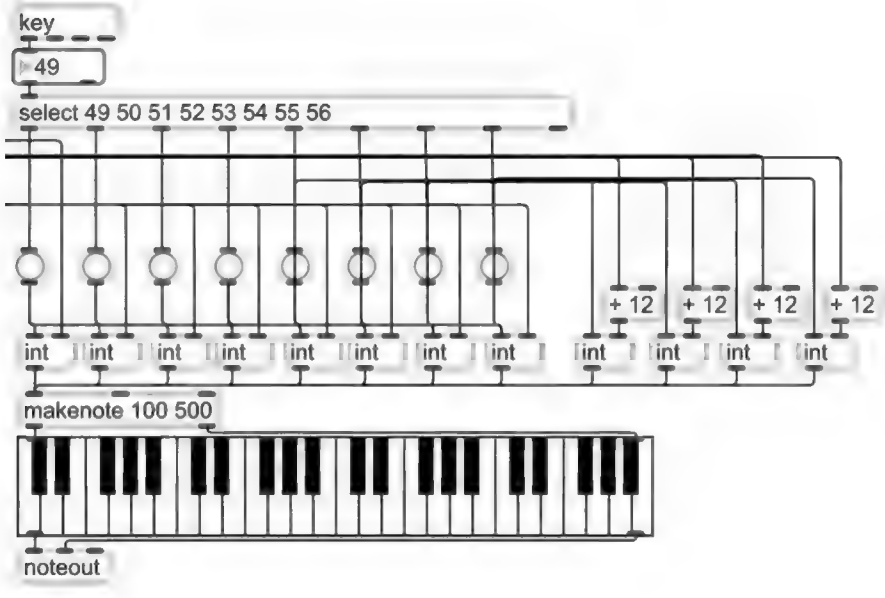


FIGURE 4.9

number keys cause ints to release stored scale degrees | chord_maker .maxpat

55. Lock the patch and press the number keys *1–8* on your computer keyboard to cause chords to play

Now your users have the option to use number keys to play specific chord functions within a key. Your program could be a great tool to explain diatonic chord functions within a given key; it even provides a graphical keyboard representation of the notes. A great addition would be to somehow display these notes on the grand staff. There is an object just for that called *nslider*. Unlock your patch and

56. Create a new object called *nslider*
57. Highlight the *nslider* and open the Inspector
58. Change the Display Mode to *Polyphonic*

Like *kslider*, *nslider* receives pitch in its first inlet and velocity in its last inlet. Let's send the note data from *kslider* to *nslider*.

59. Connect the 2 outlets of *kslider* to the 2 inlets of *nslider*, respectively

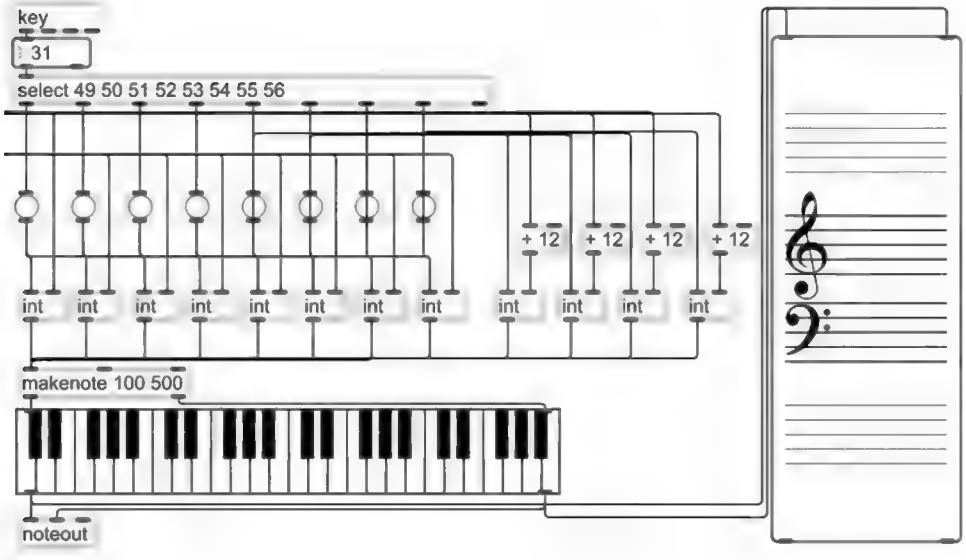


FIGURE 4.10
scale displayed in nslider | chord_maker.maxpat

60. Lock the patch and press the number keys *1–8* on your computer keyboard to see the chord tones displayed in both *kslider* and *nslider*

Now, your patch has two types of graphical representations of the notes. What about a third that displays the name of each note in the triad when played? Get ready for this trick.

Comment boxes, as you know, are very important for documenting your work and providing instructions to users. Let's type the pitch letters into separate *comment* boxes. Unlock your patch and

61. Create 7 *comment* boxes. Type the letters *C, D, E, F, G, A,* and *B* into each box, respectively

62. Highlight these comment boxes and open the Inspector
63. Change the text color to white

The *kslider* has white keys, so if we move these *comment*s with white letters on top of the keys corresponding to the letter names in the *comment*s, the user wouldn't even know the *comment*s are there until the *kslider* keys are highlighted (played). The white letters of the *comment*s would then stand out against the color of the highlighted keys. At this point, we will resize the *kslider* by clicking its bottom right corner and dragging it to the right so that we can comfortably fit the *comment*s over each key.

64. Resize the *kslider* to a larger size so that a single character *comment* box could fit comfortably over a single key
65. Move the 7 *comment* boxes over the corresponding keys of *kslider*. For example, the *comment C* should go over the *kslider* key of *C*.
66. Create 5 more *comment* boxes. Type the letters *C♯, D♯, F♯, G♯,* and *A♯* into each box, respectively.
67. Highlight these *comment* boxes and open the *Inspector*
68. In the *Inspector, c*hange the text color of the *comment*s to black
69. Move these 5 *comment* boxes over the corresponding keys of *kslider* (depending on the size of your *kslider*, you may have to change the font size of the *comments* in the *Inspector*)

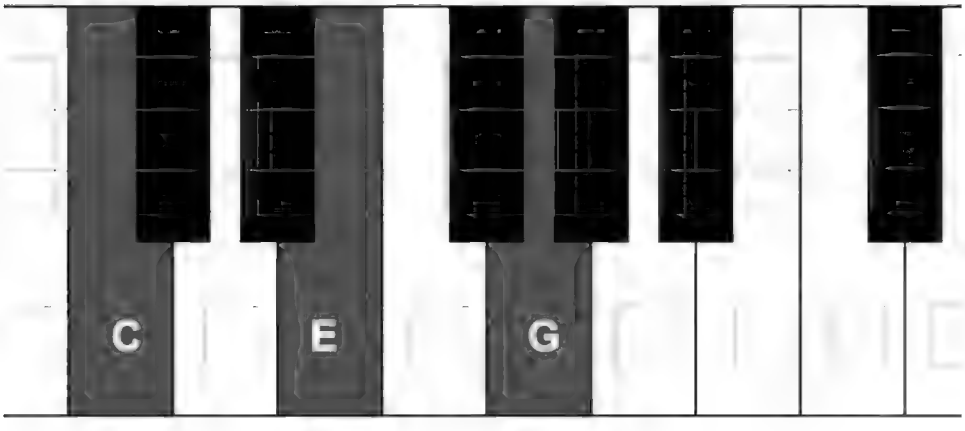


FIGURE 4.11

comment boxes overlaying kslider | chord_maker .maxpat

70. Lock the patch and press the number keys *1–8* on your computer keyboard to see the chord tones displayed in the *kslider* with letter names over them

You can feel free to copy and paste the existing *comments* to cover the other notes on the *kslider*. If your *kslider* is covering the *comment* boxes instead of the other way around, ctrl+click (Mac) or right click (Windows) the *kslider* and select "Send to Back" to move the *kslider* to the visual background of the patch.

You may also use the key command ctrl+shift+b. Note that you also have the option to "Bring to Front" an object.

It's time to add some of these files to the Presentation mode.

71. Highlight both *kslider* objects, *nslider*, the *message 2 2 1 2 2 2 1*, the 8 *button*s, and all of the *comment* boxes
72. Ctrl+click (Mac) or right click (Windows) one of the highlighted objects and select "Add to Presentation" from the contextual menu
73. Open the *Patcher Inspector*
74. In the *Patcher Inspector,* check the box marked next to "Open In Presentation" to allow your patch to open in Presentation mode by default
75. Enter a title for your patch next to *Title* such as *Chord Maker*

When your patch loads, users will have to click on the *message 2 2 1 2 2 2 1* and then enter a key on their MIDI keyboard to display a major scale in the first *kslider* using the note played on the MIDI keyboard as the tonic. Once these steps have been completed, users will then be able to play diatonic chords by pressing the number keys *1–8* on the computer's ASCII keyboard.

When designing a program, it is sometimes helpful for your patch to be ready for use with some default arguments when it loads so that the user can start making sound right away. Even if users decide they want to change the tonic and mode later, the patch should open with some basic functionality in place.

Presently, the patch requires that two steps be completed for it to work: click in the *message 2 2 1 2 2 2 1* and enter a MIDI note to act as the tonic. We can use an object called *loadbang* to send a bang to the *message 2 2 1 2 2 2 1* when the patch loads. This means the user won't have to press the *message* box as it will receive a "bang" when we "load" the patch. In addition, we can use the object *loadmess* with an argument *60* (MIDI note middle C) to send a *message 60* to the *number* box receiving from *notein* when the patch loads. Like specifying default arguments for objects, *loadbang* and *loadmess* objects are great ways to foolproof your patch so that it works properly when opened.

76. Create a new object called *loadbang*
77. Connect the outlet of *loadbang* to the *message 2 2 1 2 2 2 1*
78. Create a new object called *loadmess* with the argument *60*
79. Connect the outlet of *loadmess* to the inlet of the *number* box receiving from *notein* (Step 3)

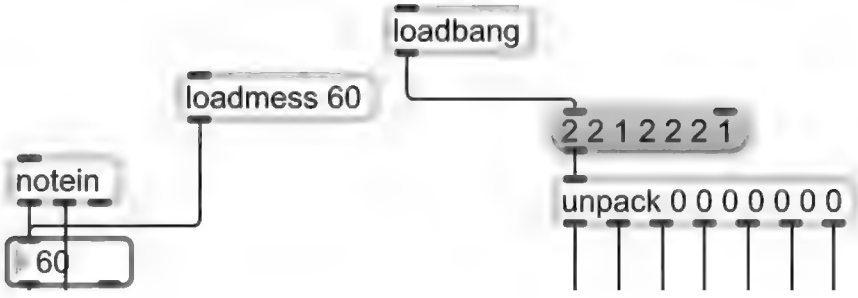


FIGURE 4.12
loadbang and loadmess trigger default actions when patch opens | chord_maker.maxpat

80. Save your patch and close it
81. Reopen the same *chord_maker.maxpat* file

Your patch will load in Presentation mode, and since the default tonic and mode have already been loaded (via *loadmess* and *loadbang*), you can press the number keys 1–8 to play the diatonic chord functions from the key of C Major.

Note that the *kslider* will display the notes of the C Major scale because of the *loadmess 60*. However, since there is no "note off" message for the *60* sent by *loadmess*, it will not disappear until it receives a velocity 0 message when you play and release middle C on your MIDI keyboard. I assume that this constant display of the scale is desirable as you would likely want to play and represent chord functions in the context of a scale. However, if it is not desirable, you can simply send a *message* box containing the text *clear* to *kslider* to release any highlighted notes.[2] Remember that this first *kslider* is not connected to your *noteout*, so you won't actually hear any notes play regardless of whether you prefer C Major to be displayed or not. The C Major scale will still load in the patch by default as a result of the *loadmess* and *loadbang* objects.

Order of Operations

In the example just given, our "load" objects need to be executed in the proper order or they won't work as they should. If the *loadmess 60* is triggered before the *loadbang*, the scale won't be built correctly since the proper arguments for the + objects (*2 2 1 2 2 2 1*) will not yet have been received. This is a good time to speak about the order of operations in Max.

In Max, all calculations set to occur at the same time do so. However, even if the objects are set to perform their tasks simultaneously, there is still an order in which objects are executed that depends upon their location in the patch. The ordering of execution is from right to left, bottom to top. In cases like the one just discussed, the order of operations matters. We will now examine one way to ensure that data get to where you want it to be at the right time and in

2. You can even connect a *loadbang* to the *message clear* if you don't want to display the default scale in the *kslider*, or remove the *kslider* from Presentation mode altogether.

the right order. Before we do so, let's do a little tweaking to the way our patch handles MIDI.

Go to *File>Save As* and save this patch as *chord_maker2.maxpat*. The current patch works well, but I really don't like the limited note duration caused by using *makenote*. The problem, once again, is that since we are generating pitches mathematically, we need some way to keep track of what pitches we played and send the appropriate velocity 0 (note off) messages for them in order to avoid "stuck notes." One such object, called *flush*, does exactly that.

Flush keeps track of pitches in its first inlet and velocities in its second inlet. When *flush* receives a bang in its left inlet, it sends out velocity zero messages for each note it received. *Flush,* however, requires a velocity message to be paired with a pitch for it to work. Previously, *makenote* did this for us by pairing its default velocity value with our generated pitches, but now, we will use the *pack* object to pair a velocity value with a pitch value. Unlock your patch, exit Presentation mode, and

82. Double click the *makenote 100 500* object and replace the text with the name of a new object called *pack*. Give *pack* the arguments *0* and *100*

Pack formats data into a list supplying an inlet for each argument given to the object. In our patch, we will use *pack 0 100* to receive, in its first inlet, all of the chord tones sent by the *int* objects. The second inlet of *pack 0 100* will be left open. Since we have given *pack* the default value *100* as its second argument, each number entering *pack* will replace the first default argument *0* and be paired with the second default argument *100*. In other words, all pitches sent to *pack* will all be paired with velocity values of *100*. Since we now have pitch and velocity pairs to work with, we can use *flush* to handle sending the appropriate "note off" messages when asked.

As you'll recall, *flush* takes pitch values in its first inlet and velocity values in its second inlet, but the output of *pack* is a single list containing both of these values. What we need is a way to separate the pitch and velocity values, now that we've paired them successfully, so that they can be received by *flush*. We will use *unpack* for this.

83. Create a new object called *unpack* (by default, *unpack* has 2 outlets)
84. Disconnect the patch cord from *pack 0 100*'s outlet to *kslider*'s inlets
85. Position *unpack* underneath *pack 0 100* and connect the outlet of *pack 0 100* to the inlet of *unpack*
86. Create a new object called *flush*
87. Position *flush* underneath *unpack* and connect both outlets of *unpack* to both inlets of *flush,* respectively

88. Connect both outlets of *flush* to both inlets of *kslider,* respectively
89. Create a *button*
90. Connect the outlet of the *button* to the first inlet of *flush*

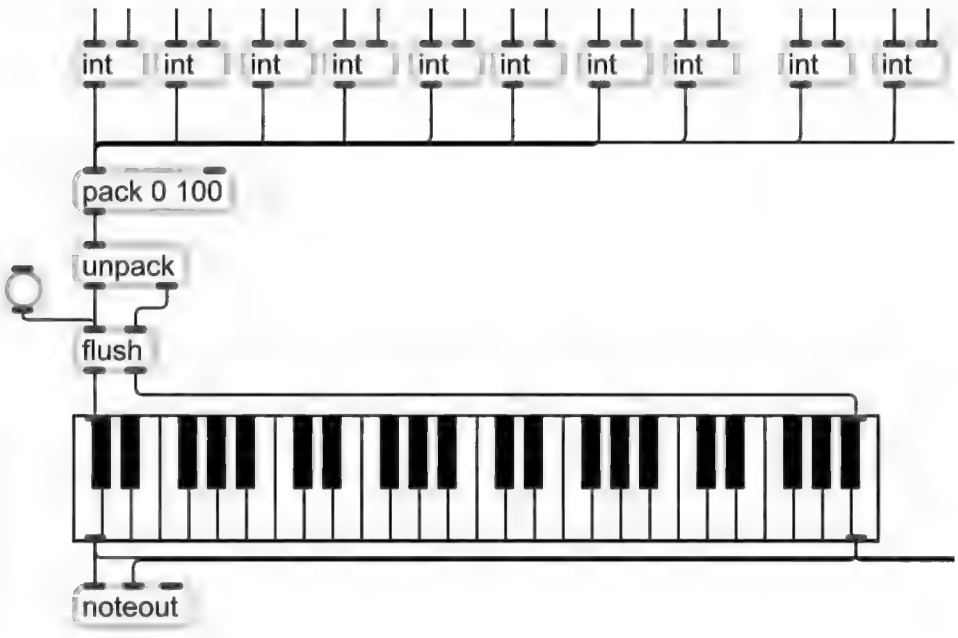


FIGURE 4.13

pitches stored in int objects are packed with a velocity 100 message | chord_maker2.maxpat

91. Lock your patch and press a number key on your ASCII keyboard. When you want the chord to stop sustaining, simply press the *button* connected to *flush* to send the proper "note off" messages

Our patch has certainly grown, but I'm still not convinced that this most recent improvement made the program any better. For starters, it's annoying to have to click that *button* every time you want the chord to stop sustaining. It would be great to set our patch so that each time we press a number key to trigger a chord, a bang was first sent to the *flush* object to stop notes from sustaining. This ordering of events can be accomplished with the *trigger* object.

The *trigger* object, or simply *t*, takes data in its inlet and sends out formatted versions of the data, dictated by the supplied arguments, out of its outlets in the order right to left. For example, if we give *trigger* the arguments *i* for integer and *b* for bang, when *trigger* receives an integer in its inlet, it will first send out a bang from its last outlet, and then send the integer out of its first outlet. Each of the outlets are labeled to match the type of arguments supplied to *trigger*. Unlock your patch and

92. Create a new object called *trigger* with the arguments *i* and *b*
93. Disconnect the first outlet of the *number* box receiving from *key* from the left inlet of *select 49 50 51 52 53 54 55 56*

94. Position *trigger i b* between the *number* box and *select* and connect the first outlet of the *number* box to the inlet of *trigger i b*
95. Connect the first outlet of *trigger i b* to the inlet of *select*
96. Connect the second outlet of *trigger i b* to the inlet of the *button* connected to *flush*

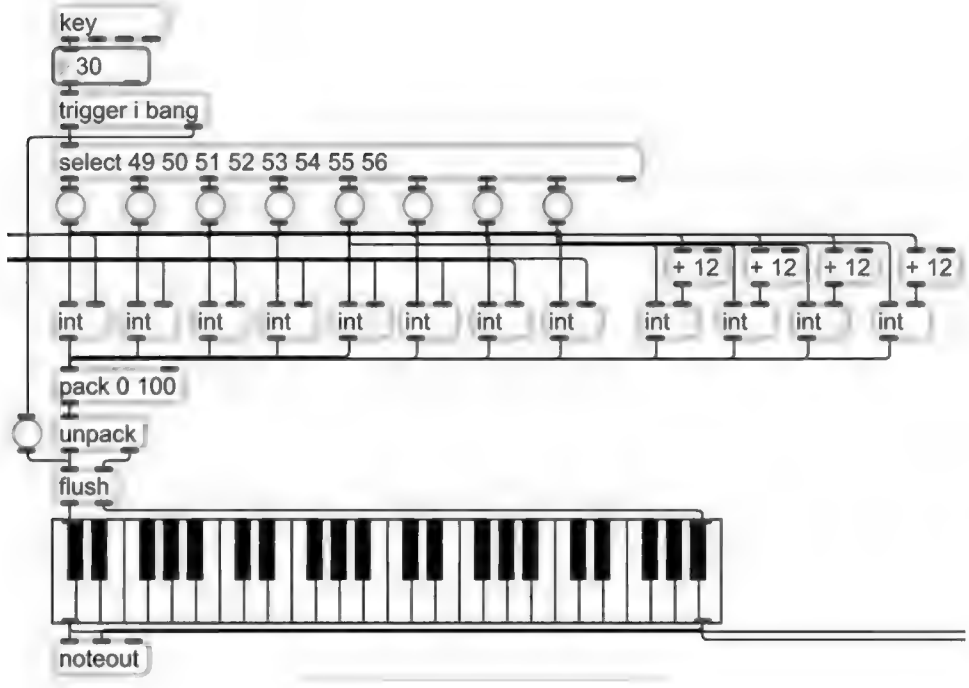


FIGURE 4.14

trigger object bangs note-off messages from flush object | chord_maker2.maxpat

When you press a number key on your ASCII keyboard, the *key* object will send the ASCII number to the *trigger* object. When *trigger* receives the number, it will first send a bang out of its last outlet (created by the argument *b*) and then, since the number is an integer, send the number out its first outlet (created by the argument *i*). The bang sent from *trigger* will cause *flush* to output the note off messages for all pitches previously received. This makes chord playback legato. Any key pressed on your ASCII keyboard will be sent to *trigger*, so even keys that are unrecognized by one of *select*'s arguments, like the space bar, can be used to stop notes from sustaining.

97. Lock your patch and press a number key on your ASCII keyboard. To stop notes from sustaining, hit any key except number keys 1–8 (the space bar is a good choice since it's the biggest key)

Save and close this patch.

Setting Values

In the previous exercise, it was necessary for us to store numbers in an *int* in order to retrieve them later. There are other objects that operate similarly to *int* including *value* and *pvar*. One issue that we overcame by using *int* was to stop notes in a *number* box from being outputted when they are changed; we did this by storing them and later manually selecting which specific notes we wanted to sound. Many objects, including the *number* object, have the ability to *set* a number inside the object without causing any numbers to be outputted. Create a new patch and

1. Create 6 *number* boxes aligned in 2 rows and 3 columns
2. Connect the first outlet of each upper *number* box in each column to the inlet of the *number* box directly beneath it
3. Create a *makenote* object with the arguments *100* and *300*
4. Connect the left outlet of each lower *number* box to the first inlet of *makenote 100 300*
5. Create a *noteout* object
6. Connect both outlets of *makenote 100 300* to the first 2 inlets of *noteout*

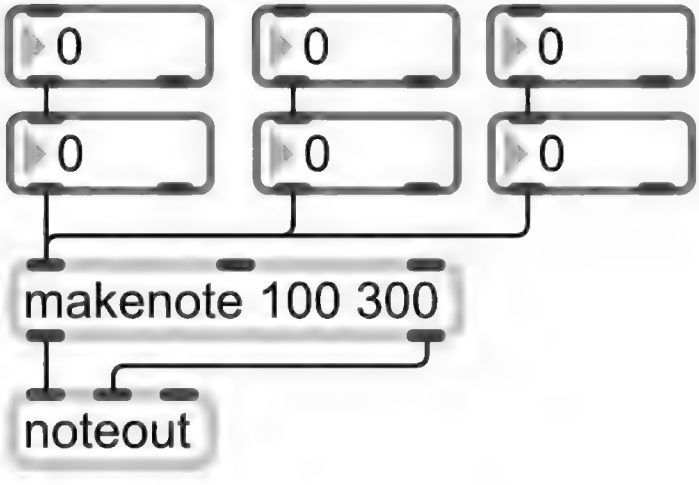


FIGURE 4.15
number boxes connected to makenote | chord _maker_quick.maxpat

As you already know, if you lock your patch and change the value in one of the *number* boxes, the number is sent to the *number* box beneath it where it is paired with a velocity in the *makenote* object and eventually sent to the *noteout* where it is synthesized.

7. Lock your patch and change the numbers in the upper *number* boxes. You will see the numbers directly beneath them change to the same values

This is typical of the data-flow in Max. However, a *message* box containing the word *set* followed by a number—for example, *set 55*—sent to a *number* box

would send the number *55* to the *number* box without causing any output to come from the *number* box's outlet. It simply "sets" the number in the *number* box. Unlock your patch and

8. Create a *message* box containing the text *set 55*
9. Connect the outlet of the *message set 55* to the inlet of one of the upper *number* boxes

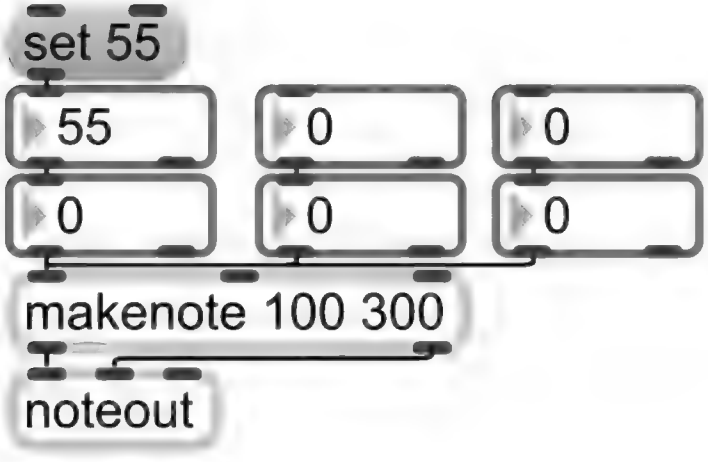


FIGURE 4.16
set number values in number box without causing output | chord_maker_quick.maxpat

10. Lock your patch and click on the *message set 55* to see the number *55* set in the upper *number* box, without being passed to the *number* box below

With the *set* message, the number *55* is simply "set" in place. Now hang on; this next part might get a little intense.

The argument $ (no, sadly, this has nothing to do with money) is a placeholder for incoming data. A number next to it—for example, *$1*—specifies which item in the incoming list of data you want the $ to hold the place for. A *message* box containing *set $1* would allow a user to send a number from a connected *number* box—for example, *55*—to the *message* box's left inlet. The number *55* would fall into the place held by the $1 and the resulting message sent from the *message* box's outlet would be *set 55*. Using the $1 would allow you to keep sending different numbers to the *message* box while preserving the proper formatting: *set* then *number*. Whew!

Let's prove this point by putting the *message* box containing *set $1* in between the output of the *+ 0* object and the input of the *number* box. Unlock your patch and

11. Disconnect the patch cord connecting the outlet of each of the 3 upper *number* objects and the inlet of each of the 3 *number* boxes directly beneath them by highlighting the patch cord and pressing the delete key

12. Create 3 new *message* boxes with the contents *set $1* and position them between each upper *number* box and the *number* box directly beneath it
13. Connect the outlet of each upper *number* box to the first inlet of each *message set $1* box directly beneath it
14. Connect the outlet of each *message set $1* box to the inlet of each lower *number* box

If you lock your patch and change the numbers in any of the upper *number* boxes, the numbers will be *set* in the *number* boxes directly beneath them, but no output will be sent to the *makenote 100 300*. Unlock your patch and

15. Create a *button*
16. Connect the outlet of the *button* to the inlet of the lower 3 *number boxes*

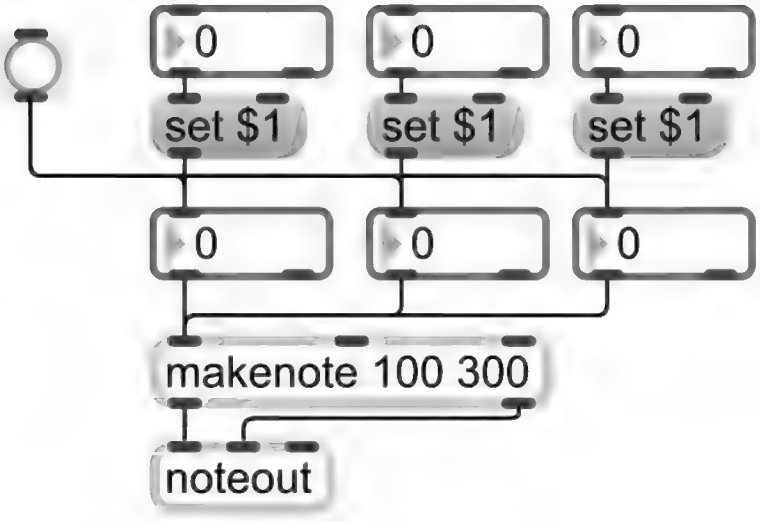


FIGURE 4.17

send bang to number boxes to pass the set numbers to makenote | chord_maker_quick .maxpat

17. Lock your patch and set the numbers in each of the upper 3 *number* boxes to a chord; it will sound only when the *button* causes the numbers to be sent to *makenote 100 300*
18. Enter the numbers 60, 64, and 67 (MIDI notes C, E, and G) in the upper 3 *number* boxes, respectively
19. Press the *button* to play the C Major triad

Using the *set* message—for those objects that permit it, like *number*—is one way that data can be manipulated in Max without causing an output. It allows you to prepare numbers, or notes represented by numbers, but refrain from causing any sort of output until a determined time.[3]

3. Note the object *pak* (not *pack*) with the arguments *set* and *0* could also be used to format a *set* message by taking a number in its second inlet. Unlike *pack*, the *pak* object sends out data when it is received in any inlet.

Now, suppose that, instead of manually entering the numbers for a triad as you did in this example, you played 3 notes using a MIDI keyboard, ensured that the notes played were voiced the way you wanted them in the *number* boxes, and then used a *button* to play them at will. You could conceivably set up the notes for each chord in your user's favorite pop song and allow her to simply use *buttons*, perhaps triggered by pressing keys on her ASCII keyboard, to perform the song.

You already know how to get notes into Max from a MIDI keyboard, so all we really need to learn to accomplish this is how to route each note of the chord to a separate location. An object called *poly* can allocate chord voicings to different locations. Unlock your patch and

20. Create a new object called *notein*
21. Create a new object called *kslider*
22. Connect the first 2 outlets of *notein* to the 2 inlets of *kslider*
23. Create a new object called *poly*
24. Create 3 *number* boxes
25. Connect the first 3 outlets of *poly* to each of the 3 *number* boxes, respectively

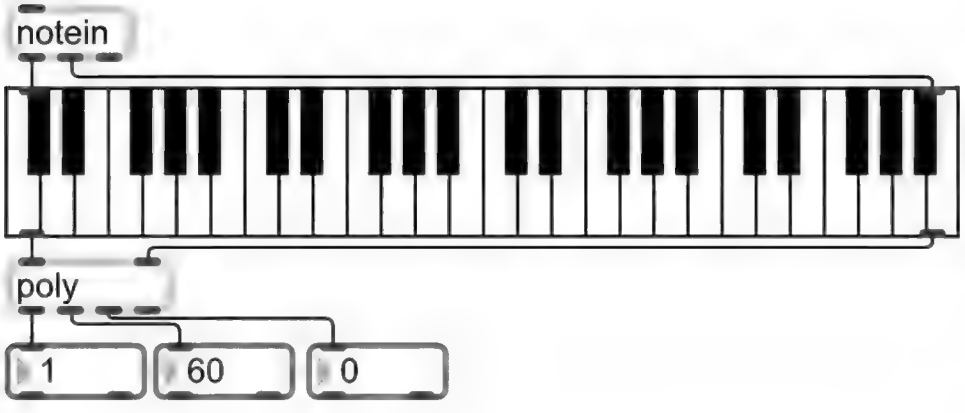


FIGURE 4.18
poly displaying values played on MIDI keyboard | chord_maker_quick .maxpat

26. Lock your patch and hold down a triad on your MIDI keyboard

Poly understands how many notes are held down as well as the pitch and velocity for each voice of the triad. *Poly*'s first outlet gives us the voice number, its second outlet gives us the pitch associated with that voice number, and the third outlet gives us the velocity. If 3 notes are held simultaneously, *poly* would label the 3 notes as voice *1, 2,* and *3*, and give the pitch and velocity for each.[4]

4. Note that since the notes are all happening at close to the same time, we will only be able to see one number at a time displayed in the *number* boxes even though there are three notes in the chord you played.

For us to visualize the entire data structure, let's *pack* the data coming from the first 3 outlets of *poly* into a list and display them in the Max window with *print*. Unlock your patch and

27. Create a new object called *pack* with the arguments *0 0 0*
28. Connect the first outlet of each of the 3 *number* boxes beneath *poly* to the 3 inlets of *pack 0 0 0*
29. Create a new object called *print*
30. Connect the outlet of *pack 0 0 0* to the inlet of *print*

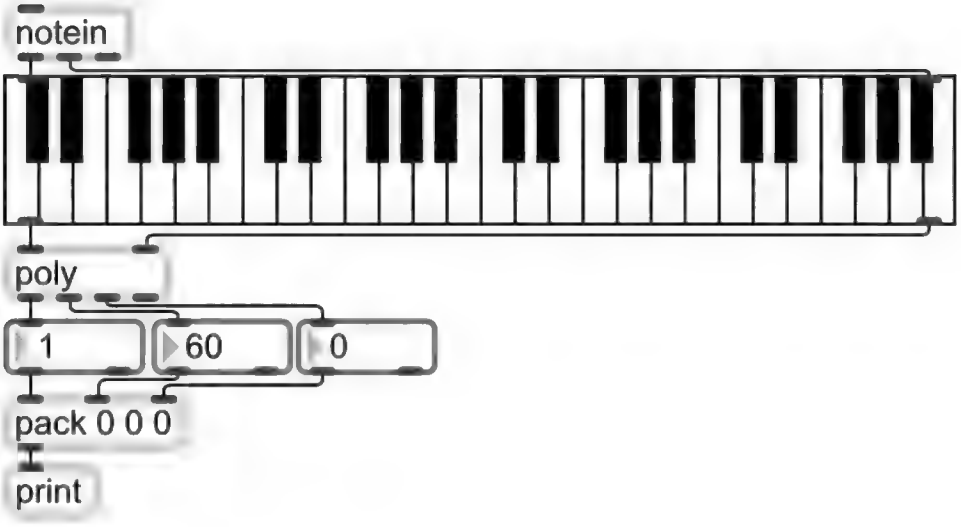


FIGURE 4.19

poly number packed into a list and displayed in floating Max window | chord_maker_quick .maxpat

31. Lock your patch and play a 3-note chord
32. Double click the *print* object to bring up the Max Window

The display at the bottom of the Max Window will show a group of 3 numbers beginning with the number 1, followed by another group beginning with the number 2, as well as a third beginning with the number 3. In each of these groups, the second and third numbers are the pitch and velocity of the notes you played. The first number is like the address, or index, of where the pitch and velocity data live—the voice number.

The goal is to set our patch so that pitch and velocity for chord voice 1 are routed to one location, while the pitch and velocity for chord voices 2 and 3 are also each routed to separate locations. To accomplish this, we can use the *route* object. The *route* object looks at the first number in a message, compares the number to the arguments given to *route* and, if the first number of the message matches one of *route*'s arguments, *route* sends out the rest of the data in the message from one of its outlets. Unlock your patch and

33. Double click the *print* object replacing the text with a new object called *route* with the arguments *1 2 3*

34. Create a new object called *print* with the argument *voice1*
35. Connect the first outlet of *route 1 2 3* to the inlet of *print voice1*
36. Create a new object called *print* with the argument *voice2*
37. Connect the second outlet of *route 1 2 3* to the inlet of *print voice2*
38. Create a new object called *print* with the argument *voice3*
39. Connect the third outlet of *route 1 2 3* to the inlet of *print voice3*

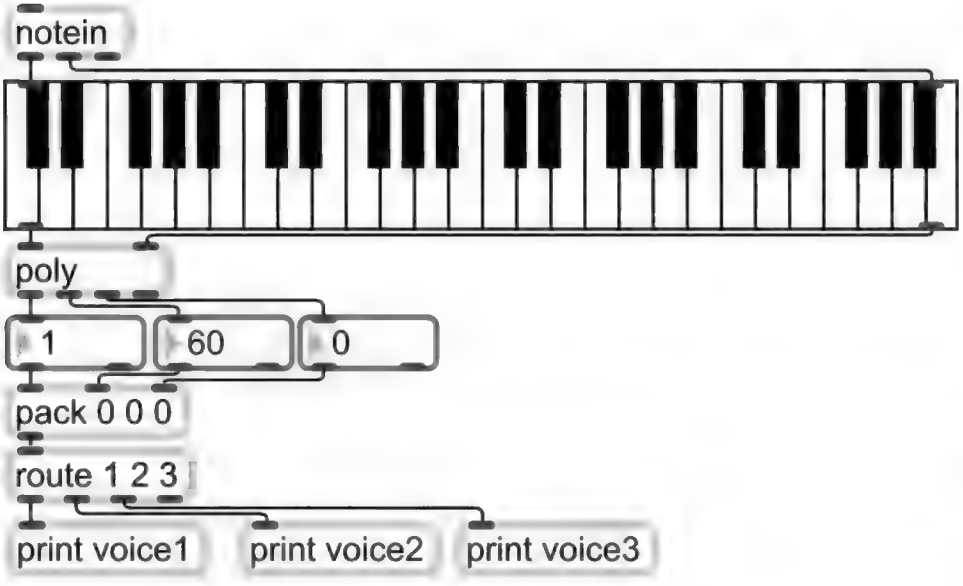


FIGURE 4.20

each voice from poly routed to a different print object | chord_maker_quick.maxpat

40. Lock your patch and double click one of the *print* objects to bring up the Max Window
41. Click the X in the bottom left corner of the Max Window to clear all of the contents
42. Play a triad on your MIDI keyboard

The voices in the triad will be sent to 3 different locations and packed into one message. Next, the message will go to *route* where the note in voice 1 will match *route*'s argument *1*. The *1* will be stripped from the message and the remaining two numbers, representing pitch and velocity of voice 1, will be sent to *print voice 1* where they will appear in the Max Window.

We have successfully isolated both pitch and velocity from each note of a triad being played. For our patch, we really only need the pitch number since we can use a *makenote* to supply a default velocity. So let's set our *pack* to forget about velocity and only work with pitch. Unlock your patch and

43. Double click the *pack 0 0 0* object and delete one of its *0*'s (changing it to read *pack 0 0* and, thus, only deal with 2 elements in a message)

Changing the number of items that *pack* can work with from 3 to 2 will disconnect the third outlet of *poly*, ultimately dropping the velocity message. *Route* will now be routing only pitch instead of both pitch and velocity.

44. Delete the 3 *print* objects as we no longer need them
45. Connect the first 3 outlets of *route* to the inlet of each of the 3 *number* boxes connected to a *set $1* message, respectively

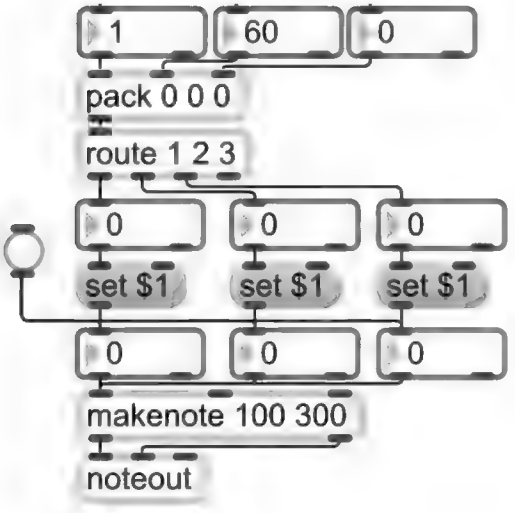


FIGURE 4.21

notes played are stored without output and triggered by button | chord_maker_quick .maxpat

46. Lock your patch and play a triad on your MIDI keyboard (it will not make a sound)
47. Click the *button* to play back the chord when desired

This is a much better alternative to manually typing in each MIDI number in the triad. Imagine making a similar patch with the multiple chords of your choosing, each able to be triggered by a *button*. In the chapters ahead, we will examine easier ways to implement chords and make our own musical instruments using these same concepts. Click *File>Save* and save this patch as *chord_maker_quick.maxpat*.

New Objects Learned:

- *unpack*
- *int*
- *nslider*
- *loadbang*
- *loadmess*
- *flush*
- *pack*
- *pak*
- *trigger* or *t*

- *value*
- *pvar*
- *set $1*
- *route*

Remember:

- Although everything in Max happens at once, there is an ordering in the way that objects are triggered depending upon where they are positioned within the patch: right to left, bottom to top.
- A *message* box containing *$1* is a placeholder for some number. The $1 is typically seen with some text in front of it. For example, *set $1* is common.

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Message Ordering—Debugging program flow
  - Note Management—Generating and managing note events

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows you to perform, in some way, in a key. Give your patch the ability to easily change keys. Use your knowledge of UI objects as well MIDI input objects as you devise your instrument.



Interactive Ear Training

In this chapter, we will review some of the objects and concepts we've learned so far and make an interactive ear-training program. We will build our ear-training program using primarily objects you already know.

In designing a program with a specific purpose, like an ear-training program, the worst technique you can take is to begin by arbitrarily inserting objects into a patch. Instead, it's best to conceptualize the task at hand and "divide and conquer." Divide the task into smaller steps and think through what actually needs to get done to accomplish the goal of the patch.

Ear Trainer

Begin a new blank patch we will use to build an ear-training program in which diatonic and chromatic intervals are randomly performed by the patch. Users will have to correctly select the name of the interval they heard from a menu. The software will give immediate feedback based on their response. To begin, create a new patch and

1. Create a *button*
2. Create 2 new object boxes both called *random* with the argument *12*
3. Position the objects next to each other horizontally
4. Connect the *button* to the first inlet of each *random 12* object

5. Create 2 *number* boxes beneath each *random 12* object
6. Connect the outlet of each *random 12* object to the *number* box beneath it

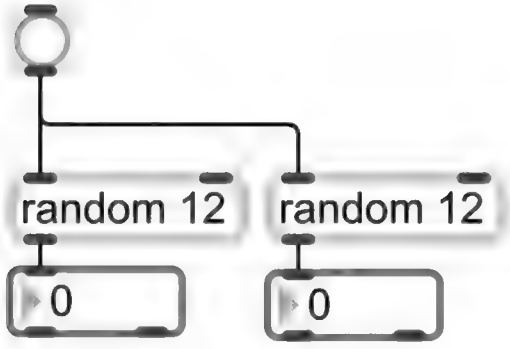


FIGURE 5.1
button triggering two random numbers between 0 and 11 / ear_training.maxpat

7. Lock the patch and click the *button* to generates 2 random numbers between 0 and 11

We will use these two random numbers to build our intervals. The *random* number on the left will be our starting pitch and we'll add the second *random* number as the added interval. As you may have realized, giving the argument *12* to *random* restricts the number of pitches to 12 allowing us to keep our intervals within one octave.

Let's add the two numbers together. Unlock the patch and

8. Create a new object called + beneath the left *number* box
9. Connect the first outlet of the *number* box on the left to the first inlet of +
10. Connect the first outlet of the *number* box on the right to the second inlet of +

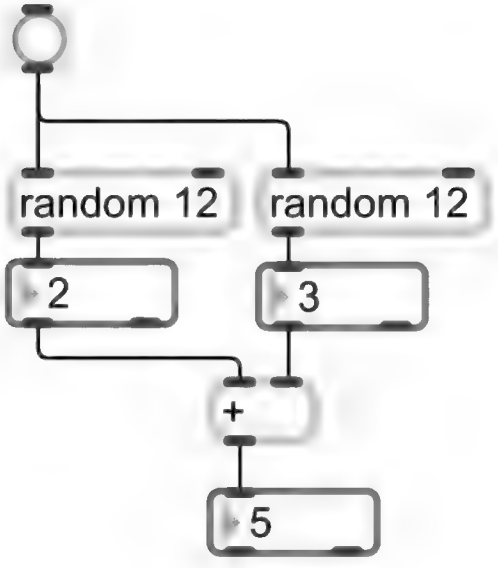


FIGURE 5.2
button adds two random numbers / ear_training .maxpat

11. Create a new *number* box beneath the + object
12. Connect the outlet of the + object to the inlet of the *number* box
13. Lock your patch and click the *button*

Note the way we've ordered the two *random 12* objects: the one on the right will supply the number as an argument through +'s second inlet, which will be added to the number received in +'s first inlet. The calculation for + does not occur until a number is received in its first inlet.

Remember the "right to left, bottom to top" order of operations discussed in the last chapter. If we ordered the *random 12* objects differently, the + would be adding a value previously received in its last inlet. If left, when a user loaded our patch and clicked the *button*, it would effectively produce two random numbers, but + would actually be adding 0, the default value, to the number received in its left inlet since it had not yet received a random number value as an argument. As you know, the + object without an argument has a default value of 0 which would be added to your random note, so your first interval would always be a *unison*. For the same reason, supplying a nonzero argument for + would also start the patch with the same interval.

We have now created the two notes to form our interval. The *number* box to the left will be the lower of the two pitches we will use to form our interval, and the *number* box receiving from the + object will be the higher of the two pitches. Adding the two random numbers to form the higher pitch in our interval as we have done is one way to ensure that we keep the larger number where we want it. If we had simply subtracted one random number from the other, it can become difficult to account for negative numbers as the difference and the uncertainty of which of the two random numbers is larger.

Remember, middle C is MIDI note *60*, so the numbers (*0–11*) we're using are quite low in pitch to be used for an ear-training patch. However, it's important that we use the argument *12* for both *random* objects to generate numbers between *0* and *11*. Using larger values as *random*'s arguments would introduce the possibility of one random number—for example, *65*—being added to another random number of *70*. This would produce a note that is way too high. Instead, we'll use just the 12 pitches at their lowest octave and transpose them up a few octaves. We know that adding *12* half steps to a note is the same as adding one octave, so let's add 48 to our notes to raise them 4 octaves. This will start our notes in the octave below middle C. Unlock your patch and

14. Create 2 new objects each called + with the argument *48*. Arrange them horizontally
15. Connect the outlet of + object you created in Step 7 to the first inlet of the newly created + *48* object on the left
16. Connect the first outlet of the *number* box receiving from the left *random 12* object to the first inlet of the other newly created + *48* object on the right

17. Create 2 *number* boxes
18. Connect the outlet of each + *48* object to the left inlet of the 2 *number* boxes, respectively

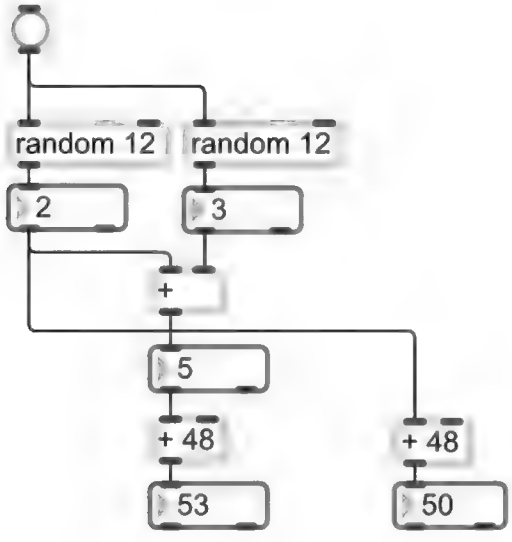


FIGURE 5.3
adding 4 octaves to both random numbers / ear_training.maxpat

19. Lock your patch and click the *button*

Our interval is now formed in a sensible octave. To determine the number of semitones beneath the notes, we'll simply subtract the lower note, that is, the *number* box on the right, from the higher note, the *number* box on the left, using the - object.

You may be wondering why the second + *48* object in the figure above is not located directly beneath the first *random 12* object. The reason, once again, has to do with the order of operations. We want to ensure that the number on the right gets to the last inlet of the - object to set the subtrahend before the number on the left is received in -'s first inlet, which triggers the calculation. The resulting number will give us a value between *0* and *12* which we can interpret as semitone differences between both notes in the interval and will be the same value as one of our random numbers (since we are simply reversing the interval formation process to ensure accuracy). This is similar to the way we used intervals with a *umenu* in Chapter 3. We can even connect a *umenu* to our present patch.

20. Create a new object called -
21. Connect the first outlet of the right *number* box to the last inlet of the - object
22. Connect the first outlet of the left *number* box to the first inlet of the - object
23. Create a *number* box
24. Connect the outlet of the - object to the inlet of the *number* box

25. Open the patch you made in Chapter 3 and copy the *umenu*, then paste it into this patch. If you can't find the patch, create a new object called *umenu*
26. Connect the outlet of the *number* box to the inlet of the *umenu*
27. Click on the *Inspector* for *umenu* and select *Edit* to the right of menu items, adding the following text: *unison, minor 2nd, Major 2nd, minor 3rd, Major 3rd, Perfect 4th, tritone, Perfect 5th, minor 6th, Major 6th, minor 7th, Major 7th, Octave*

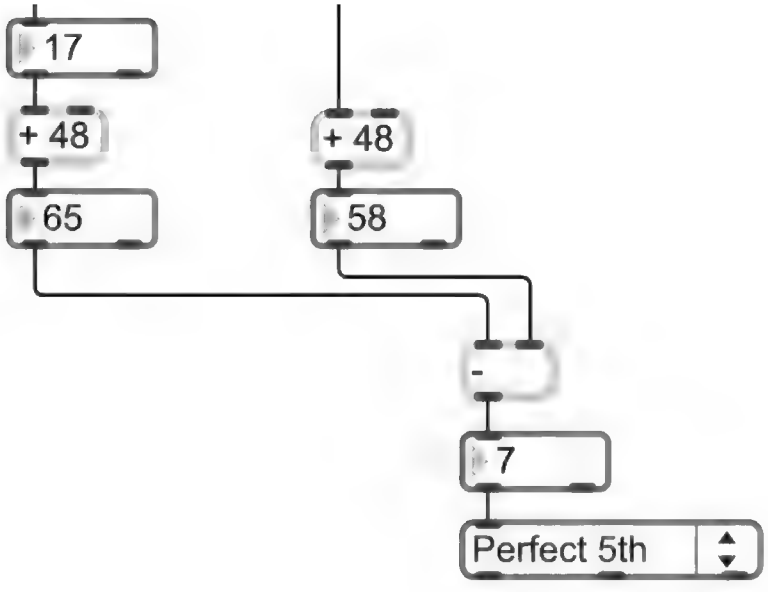


FIGURE 5.4

interval between numbers triggers umenu item / ear_training.maxpat

28. Lock your patch and click the *button*

As you trigger a random interval to be formed, it will be calculated and displayed in the *umenu*. We will use this number to check the correct interval against the interval the user selects. Let's build that "assessment" part of the program now. Unlock your patch and

29. Copy the *umenu* object you just made and paste a copy to the right of the existing *umenu*
30. Create a *number* box
31. Connect the first outlet of the newly created *umenu* to the inlet of the *number* box

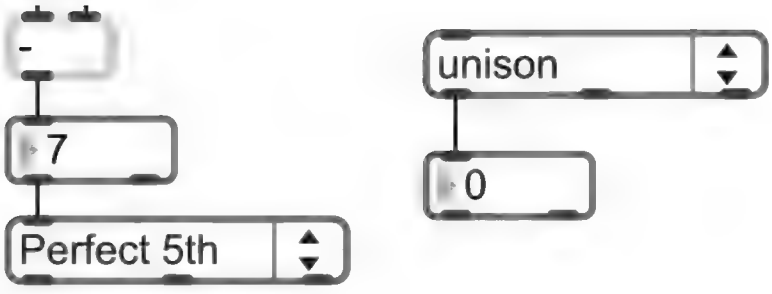


FIGURE 5.5
two umenus / ear_training.maxpat

This second *umenu* is what the users will select as their choice for the correct interval. We will need to compare the selected interval against the correct interval. For this, we can use the *select* object, taking the numerical interval as *select*'s argument.

32. Create a new object called *select*
33. Connect the first outlet of the *number* box receiving from the - object to the last inlet of the *select* object
34. Connect the first outlet of the *number* box receiving from second *umenu* to the first inlet of the *select* object

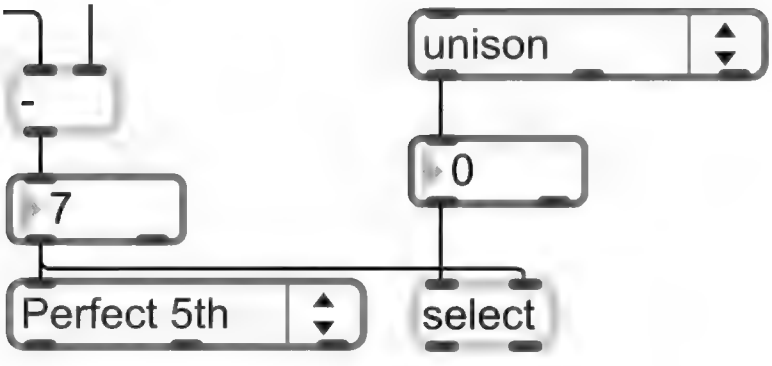


FIGURE 5.6
evaluating selected umenus / ear_training.maxpat

As in Chapter 3, the intervals listed in the *umenu* are ordered so that *unison* is item number *0*, a *minor second* is item number *1*, and so on. You may have noticed that we did not give *select* an argument to compare incoming data to. Instead, we are sending *select*'s second inlet the correct interval number so that when a user chooses an interval from the *umenu*, it will be compared against the correct interval set as *select*'s argument. If the selected interval is the same as the correct interval, *select* will send a bang out of its first outlet; otherwise, *select* will send the incorrect interval number out of its second outlet.

Let's connect two *buttons* to the outlets of *select*, so that if the user's selection is correct, the left *button* will blink, and if the selection is wrong, the right *button* will blink.

35. Create 2 *button* objects
36. Connect each of *select*'s outputs to the inlet of each *button*, respectively

37. Lock your patch and click the *button* at the top of the patch connected to the 2 *random* objects
38. After seeing the interval displayed in the first *umenu*, choose the correct interval from the other *umenu* to observe the *button* at *select*'s first outlet blink

A blinking *button* is a nice indicator for a correct or incorrect response, but the *led* object might be better suited for this patch since it allows us to control the length of the blink time. This will allow the user to see the program's feedback regarding correct or incorrect answers for a bit longer than a *button*. Unlock your patch and

39. Create 2 new objects call *led*
40. Connect the outlet of each *button* to the inlet of each *led,* respectively
41. Open the Inspector for the *led* on the left
42. Change its *Blink Time in Milliseconds* to 2000
43. Change its *Hilted Led Color* to a green color
44. Change its *Inactive Led Color* to a white color
45. Check the box next to *Ignore Click*
46. Close the *Inspector*
47. Open the *Inspector* for the *led* on the right
48. Change its *Blink Time in Milliseconds* to 2000
49. Change its *Inactive Led Color* to a white color
50. Check the box next to *Ignore Click*
51. Close the *Inspector*

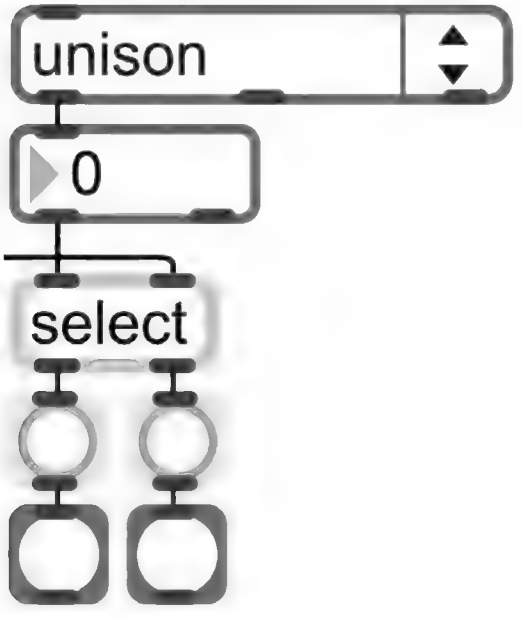


FIGURE 5.7

umenu determines correct or incorrect selection / ear_training.maxpat

52. Lock your patch and click the *button* at the top of the patch connected to the 2 *random* objects
53. After seeing the interval displayed in the first *umenu*, choose the correct interval from the other *umenu* to observe the *led* object at *select*'s first outlet blink

As you can see, a correct response will cause the *led* to blink green for two seconds. An incorrect response will cause the other *led* will blink red for two seconds. Checking *Ignore Click* in the *Inspector* means that these *led* objects will no longer blink when they're clicked, only when an answer is received.

At this point, as you have realized, the patch doesn't have any objects to allow us to hear the interval in question. That's an easy fix, as you know, but let's also build in the ability to replay the interval in case our users would like to hear the interval more than once before selecting an answer. Unlock your patch and

54. Create 2 new objects call *int*
55. Connect the first outlet of each *number* box receiving from the + *48* object (Step 14) to the left inlet of the *int* objects, respectively

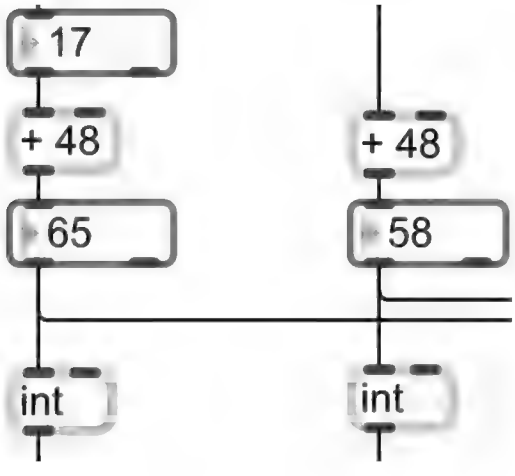


FIGURE 5.8
pitches stored in int / ear_training.maxpat

As you'll recall, a number sent to the second inlet of *int* stores the number without causing any output. A number sent to the first inlet will store the number and cause the number to pass through *int*'s outlet immediately to any objects it's connected to (like the synthesis objects we'll add in a moment). *Int* takes a bang from a *button* in its first inlet to cause the stored notes to be sent out again. If we connect a *button* to each *int*, our users can replay the sounding interval before choosing their answer from the *umenu*.

56. Create a *button*
57. Connect the outlet of the *button* to the left inlet of both *int* objects
58. Create a *makenote* object with the arguments *100 500*

59. Connect the outlet of each *int* object to the first inlet of *makenote 100 500*
60. Create a *kslider* object
61. Connect both outlets of *makenote 100 500* to both inlets of *kslider,* respectively
62. Connect both outlets of *kslider* to the first 2 inlets of *noteout,* respectively

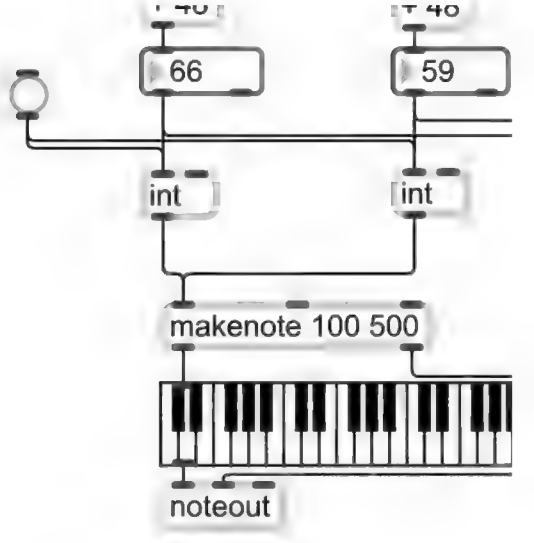


FIGURE 5.9

synthesizing stored numbers by sending them to makenote via button / ear_training.maxpat

63. Lock your patch and click the *button* at the top of the patch connected to the 2 *random* objects
64. After hearing the interval played, choose the correct interval from the other *umenu* to observe the *led* object at *select*'s first outlet blink

The program is now working fine and could be a great way to generate random intervals harmonically. If you'd like to generate intervals melodically, that is, one after the other, we can easily do so with an object called *pipe*.

The *pipe* object receives data in its left inlet and delays the output of the data by some number of milliseconds specified as an argument. For example, if we specify the argument *300* for *pipe* and sent a bang from a *button* to *pipe*'s first inlet, the bang would be delayed 300 milliseconds before it came out of *pipe*'s outlet.

If we connected a *button* to the lower pitched *int* and used a *pipe* to delay that same *button* by *300* milliseconds before it sent a bang to the higher pitched *int*, when a user clicked the *button,* a bang would hit the lower pitched *int* and cause it to sound; then, 300 milliseconds later, the bang would make its way through *pipe* and hit the higher pitched *int*. Let's build it. Unlock your patch and

65. Create a new object called *pipe* with the argument *300*
66. Create a *button* above *pipe 300*

67. Connect the outlet of the *button* to *pipe 300*'s left inlet
68. Connect the outlet of the same *button* to the *int* object on the right
69. Create a *button* beneath *pipe 300*
70. Connect the outlet of *pipe 300* to the inlet of the *button*
71. Connect the outlet of the *button* to the *int* object on the left

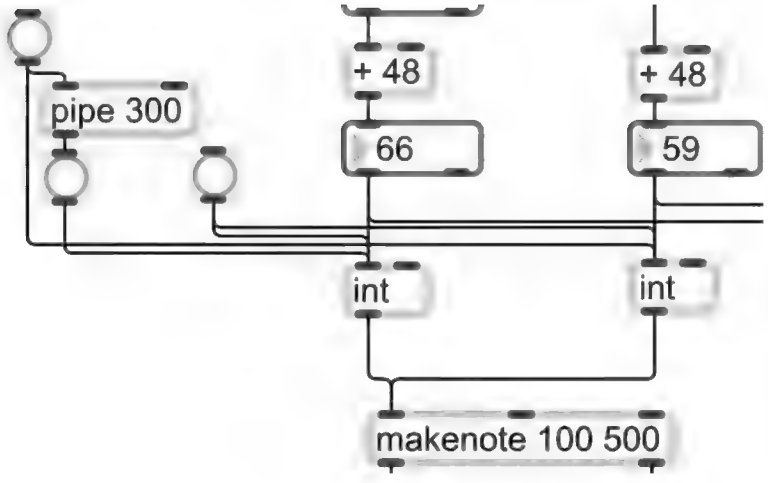


FIGURE 5.10
delaying the playback of one note in the interval / ear_training.maxpat

72. Lock your patch and click the *button* connected to *pipe* to play the interval melodically

Please note that when possible, *pipe* should be avoided as a means of getting data to the right place given Max's order of operations. Instead, use objects like *trigger* and logical object placement as we mentioned in the previous chapter. For instance, if you reverse the order of the *random 12* objects, the timing of the numbers being added will get messed up and your patch won't work correctly. This is another reason that it's important to keep your patch well organized with a logical data-flow; keep in mind that some objects, such as + and -, output data only when they are received in its first inlet.

Now, you have 3 *buttons* controlling your patch: one creates a random harmonic interval, one replays the interval harmonically, and one replays the interval melodically. Once they've heard the interval, users then select the correct answer from the *umenu*. The next step is to label the patch so your users will know how to control it. Unlock your patch and

73. Create a *comment* box containing the text "Create Interval" next to the *button* triggering the *random 12* objects
74. Create a *comment* box containing the text "Replay Interval Harmonically" next to the *button* triggering the 2 *int* objects
75. Create a *comment* box containing the text "Replay Interval Melodically" next to the *button* triggering the *pipe 300* object

76. Create a *comment* box containing the text "Correct" next to the left *led* (blinks green)
77. Create a *comment* box containing the text "Incorrect" next to the right *led* (blinks red)
78. Create a *comment* box containing the text "Choose an Interval" above the second *umenu*

It would make sense at this point to select objects and comments that you feel are important for your users to see and add them to Presentation mode.

79. Highlight all control *buttons*, both *leds*, the second *umenu*, and the *comments* made in the previous step and add them to Presentation mode
80. Open the Patcher Inspector and check the box marked "Open In Presentation"

Again, feel free to resize or reorient objects however you like in Presentation mode. You may also decide that you want to add the *kslider* to Presentation mode, and perhaps add a detailed set of instructions for your users: it's just another *comment* box.

Further Customization

As you know, you can change the background color of the patch in the *Patcher Inspector*. It might be nice, however, to put an actual image into the patch. Perhaps, you want to create a custom background for your patch that looks like your classroom, or maybe just insert an image of your students' favorite cartoon character for decoration. Inserting an image can be done with the *fpic* object. Unlock your patch and and

81. Create a new object called *fpic*

The *fpic* object is a resizable window for displaying an image. You can send *fpic* a *message* box containing the text *read* to open a dialog box by which you can select an image file to display in your patch, or you can simply open the Inspector for *fpic* and click the *Choose* button next to "Image File." In the *Chapter 06 Examples* folder in the support files for this book, you will find an image file called *somepicture.png*. Let's load this file in *fpic* to display it as our background.

82. Open the *Inspector* for *fpic* and click the *Choose* button next to "Image File"
83. Locate the file *somepicture.png* in the *Chapter 06 Examples* folder that accompanies this book[1]

1. If you do not have access to that file, you may double click the text <none> located next to *Choose* and type in *sunset.jpg* to use a default image file already embedded in Max.

84. Click on the file when you locate it, then click OK and close the *Inspector*
85. Add *fpic* to the Presentation mode
86. Enter Presentation mode
87. Click on the bottom right corner of *fpic* and resize it as you would any other UI object like *kslider* or *nslider*
88. Ctrl+click (Mac) or right click (Windows) on *fpic* and select "Send to Back" to put the *fpic* image in the background with the controls on top

Arrange your patch in such a way that the controls are clearly organized with their *comment* descriptions. With a dark background like the image I've supplied, it may be necessary to change the color of the *comment*s or to move the *comments* over a lighter part of the *fpic* image. Use your discretion.

89. Resize the patcher by clicking and dragging in the bottom right corner of the patcher window so that only the *fpic* and the control objects are visible. There should be no patcher background visible
90. Open the *Patcher Inspector* and uncheck both *Show Horizontal Scrollbar* and *Show Vertical Scrollbar* to remove all scrollbars from this patch
91. In the *Patcher Inspector*, enter *Ear Training* to the right of "Title"
92. Close the Inspector and save your patch as *ear_trainer.maxpat*

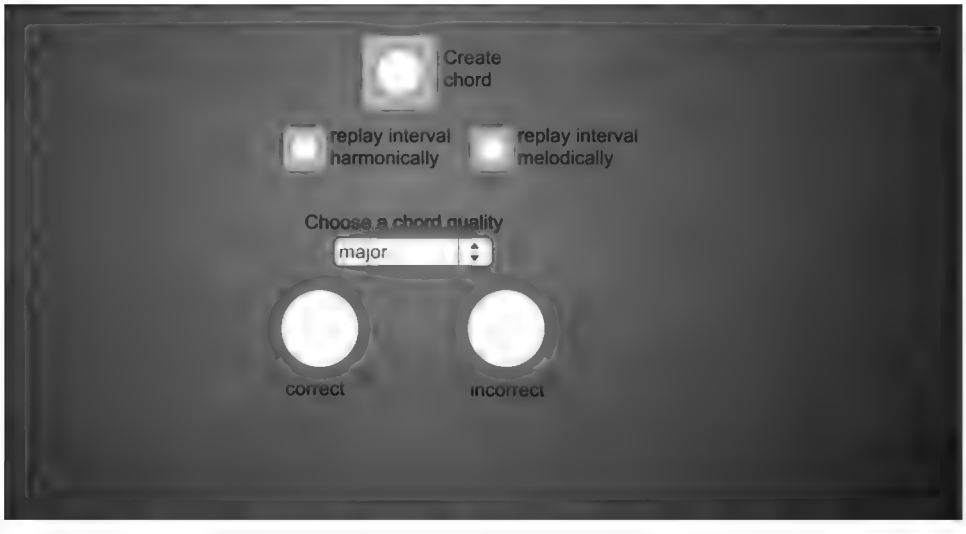


FIGURE 5.11
patch displayed in Presentation mode with fpic image as background / ear_training.maxpat

New Objects Learned:

- *led*
- *pipe*
- *fpic*

Remember:

- some objects, like + and -, output data only when it is received in a certain inlet

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create an ear-training patch like the one you just made that works with triad qualities (major, minor, augmented, and diminished) instead of intervals.



Data Structures

In this chapter, we will write a program that randomly generates diatonic pitches at a specified tempo. We will learn how to filter chromatic notes to those of a specific mode by using stored data about scales. By the end of this chapter, you will have created a patch that composes diatonic music with a simple rhythm. We will also learn about adding objects in order to expand the Max language.

For this chapter, you will need to access to the companion files for this book.

Adding External Objects

New words are added to the English language all the time. If you open a dictionary from the 1950s, you're not going to find commonly used words like *ringtone* or *spyware*. There was a need to incorporate these words into the language because they serve a specific function. In the same regard, the Max programming language can also be expanded through the development of *external objects* created by third parties.

Third party objects are often referenced in journal articles, forum posts on the Cycling '74 website (cycling74.com), and a number of repository websites for max objects. A brief, and by no means comprehensive, list of repository websites is given at the end of this chapter.

Of course, at this point, we're still learning the Max language apart from adding external objects. So why are we discussing external objects at all? Well, the beauty of Max is that you can expand the language easily to include objects that can speed up your process.

For example, in Chapter 4, we made scales and chords by following a number of steps. To speed up the process of implementing scales and chords in my own patches, I developed a set of external objects called the Modal Object Library that contains objects for quickly building chords and scales. Since you already know how to build scales and chords in Max, using these objects will speed up our process of using scales and chords in our patches instead of having to copy large chunks of objects from old patches whenever you want to implement a particular scale.

To use a third party object in Max, it must be part of the "Max search path"; that is, the third party object must either reside in the Max application folder or in another folder that you have added to the Max search path. For the latter,

1. Create a new folder on your computer (I suggest in the Documents folder) and give it a name like "My Max Objects"
2. From within Max, choose *Options* from the top menu and select *File Preferences*. You will see that Max is already searching for usable files in two locations inside the application folder: *Patches* and *Examples*
3. Click the + at the bottom left to add a search path
4. Select the *Choose* button and locate the "My Max Objects" folder you just created on your computer
5. Click OK and close the *File Preferences* window

Now, any third party objects that you download and place in this folder will be incorporated into your Max language. One of the companion files you downloaded for this book is a file called the *EAMIR SDK*. When you install it, it will place a number of patches and external objects in the Max application folder thus adding those patches and objects to the Max language as it exists on your computer. If, for some reason, the installer failed on your machine, you may unzip the *EAMIR SDK Manual Install* file and copy the folder into the "My Max Objects" folder you created. If you do both of these things, the Max window will warn you that you have duplicate objects in the Max search path.

6. Take a moment to install the *EAMIR SDK* using one of the methods described

Tonality in Max

I began using Max in 2005 when I was in graduate school. I asked myself then the same question that many of my students ask me now when I begin teaching them Max: "What do you do with this program?" In fact, at the end of a semester studying Max, I didn't use Max for quite some time because I wasn't sure if there was anything that *I* could use it for.

My interests were primarily in tonal and near tonal music, modality, and, in particular, how modal relationships can be formed between tonal centers. I

wanted to be able to give Max a tonic and a mode name and have it give me the pitch classes of that mode as well as some basic information about that mode like "What chords are formed in this mode?" and "How does this mode relate to other tonal centers?" As far as I could tell, Max had none of these features.

Sometime later, I took an advanced Max class because I wanted to give Max another chance. For my final project, I decided that I would attempt to create a patch that would give me the pitch classes of each tonic and mode similarly to the *chord_maker* patch we made in Chapter 4. That patch became what is now the *modal_change* object, the first object of what later became the Modal Object Library.

Create a new patch and

1. Create a new object called *modal_change*
2. Open up the Help file for *modal_change*

When the Help file leads, choose a tonic and mode from the attached *message* boxes or *umenus*. Notice that the *modal_change* object operates just like the *scale_maker* patch we created in Chapter 4: it takes a note as a tonic and builds a scale based on whole steps and half steps. It simply does it all in one object instead of using multiple objects.[1]

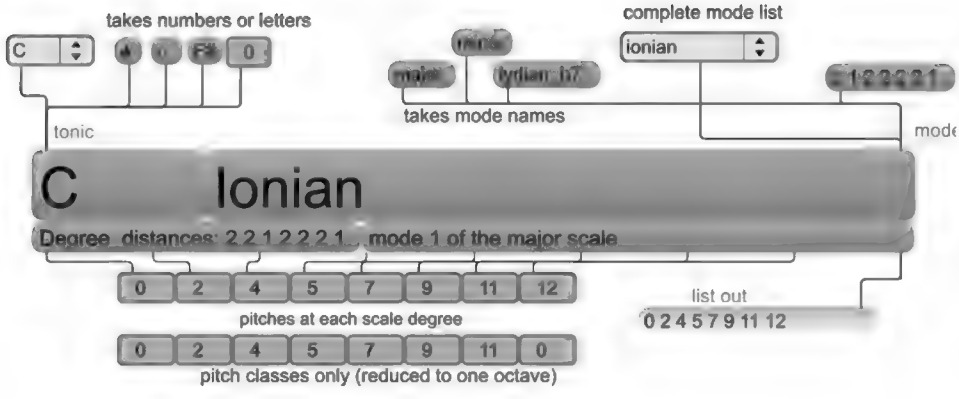


FIGURE 6.1
excerpt of modal_change Help file | modal_change .maxhelp

Filtering Chromatic Notes to Diatonic Notes

With *modal_change*, we have access to the pitch classes[2] for the 28 diatonic modes of the major, harmonic minor, melodic minor, and harmonic major (major scale with a flatted sixth scale degree) with respect to any tonic. You can also add your own "whole step/half step" pattern to create a custom scale.

A few of the patches we made early in this book were atonal. But what if we wanted to filter a chromatic note to a diatonic one? What if we want to take

1. We will use the *modal_change* object and other similar objects in the *EAMIR SDK* to speed up the patching process. You are under no obligation, however, to use these objects permanently if you don't want; you can simply build scales and chords the way we did in Chapter 4.

2. We will define a *pitch class* as the numerical equivalent of a pitch regardless of octave. All C's, regardless of octave designation, are equal to 0; all C♯'s or D♭'s are equal to 1; all D's equal 2, and so on. Pitch classes are numbered 0–11. The number 12 is a C, so it is regarded as pitch class 0.

the *random atonal trash* patch that we created in Chapter 2, and turn it into a *random tonal trash* patch? Hey, it might not be trash after this!

It turns out that there is a simple mathematical operation that you likely didn't learn in grade school called *modulo*. Modulo, or *mod*, is like division except that it outputs the remainder. For example, *60 mod 12* divides the number 60 into 5 equal parts of 12. The remainder is 0. We would then say that *60 mod 12 is equal to 0*. The expression *61 mod 12* means that 12 goes into 60 a total of 5 times with a remainder of 1. We would then say that *61 mod 12 is equal to 1*.

The interesting thing about this is that MIDI note 60 is a C. MIDI note 61 is a C♯/D♭. C is pitch class 0 and C♯ is pitch class 1. When you use modulo 12 on a MIDI pitch number, all of the C's at any octave (12, 24, 36, 48, 60, etc.) will yield a *0*. All of the MIDI C♯'s (13, 25, 37, 49, 61, etc.) will yield a *1*. All of the MIDI D's (14, 26, 38, 50, 62, etc.) will yield a 2. Thus modulo *12* will take a MIDI note number, regardless of octave, and yield a pitch class number. Imagine how interesting it would have been if your eighth grade math class had had discussions like this!

In Max, modulo is expressed with the symbol %. Close both open patches and create a new patch.

1. Create a *number* box
2. Create a new object called % with the argument *12*
3. Connect the first outlet of the *number* box to the first inlet of *% 12*
4. Create a *number* box beneath the *% 12* object
5. Connect the outlet of *% 12* object to the inlet of the *number* box beneath it

The object *% 12* will reduce MIDI notes in the *number* box to pitch classes. Let's also determine the octave designation of the pitch. To find the octave designation, simply divide the MIDI note by 12 using the / object.

6. Create a new object called / with the argument *12*
7. Connect the first outlet of the *number* box to the first inlet of */ 12*
8. Create a *number* box beneath the */ 12* object
9. Connect the outlet of */ 12* object to the inlet of the *number* box beneath it

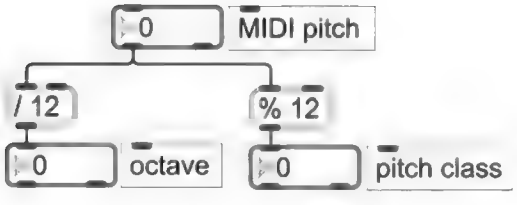


FIGURE 6.2

getting pitch class and octave designation | random_tonal_trash .maxpat

10. Lock your patch and enter the number *60* (middle C) into the *number* box

The patch reveals that the pitch class for Middle C is *0*, a C, and the octave designation is *5*, the fifth octave. Note that the MIDI octave designations are one octave lower than traditional scientific pitch notation (Young, 1939). Middle C (MIDI note 60) is at the fifth MIDI octave (C3), but at the sixth scientific pitch notation octave, C4. By opening the *Inspector* on a *number* box, you can change the *Display Format* to see the numbers in a *number* box displayed as *MIDI* notes or *MIDI C-4* notes, the latter of which shows MIDI note 60 at C4.

Random Tonal Music

11. Create a new object called *random* with the argument *128*
12. Create a *button*
13. Connect the outlet of the *button* to the first inlet of *random 128*
14. Connect the outlet of *random 128* to the inlet of the *number* box you created in Step 1

This is similar to the way you started to build the *rat_patch* before.

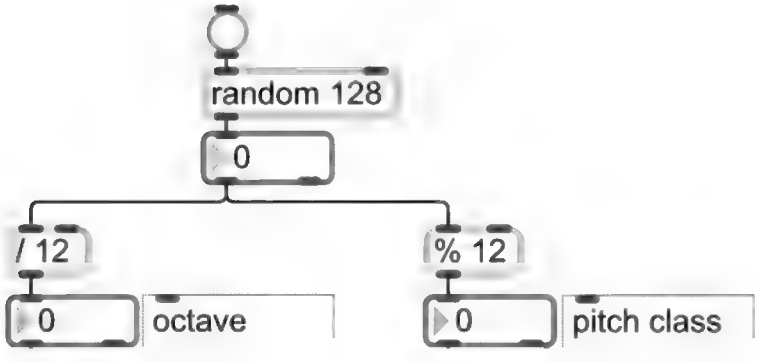


FIGURE 6.3
pitch class and octave designation from a random number | random_tonal_trash .maxpat

15. Lock your patch and click the *button* to generate random notes

We're going to make this patch generate random notes in the key of C Major, but let's first take a moment to discuss one way that Max can store data about the pitches in C Major using an object called *coll,* short for *coll*ection. Unlock the patch and

16. Create a new object called *coll* and give it the argument *pitches*
17. Connect the first outlet of the *number* box connected to % *12* (Step 4) to the inlet of *coll pitches*
18. Create a *number* box
19. Connect the first outlet of *coll pitches* to the inlet of the newly created *number* box

In this instance, the *pitches* argument we gave *coll* is actually a made-up name that we will be using to distinguish this *coll* object from other ones. This will make more sense in a few moments.

20. Lock the patch and double click the *coll* object

A blank text window will pop up: this is the heart of the *coll* object. The text window is, in essence, a list of data values stored at different indexes in the following format:

index number, stored data;

The index number is the address where the data live. If we were to connect a *number* to *coll*'s first inlet, whatever number we entered into that *number* box would be interpreted by *coll* as an index number. *Coll* would then determine the stored data, called an element, at that index number and send the data out its outlet.

21. The first index we'll type in the *coll* window is the number *0* with the element *0*. Format it the following way separating the index and the element with a comma and a space and ending the line with a semicolon:

0, 0;

22. Next, add the following lines directly beneath the one you just entered so that the contents of the *coll* window look like this:

0, 0;
1, 0;
2, 2;
3, 4;
4, 4;
5, 5;
6, 5;
7, 7;
8, 7;
9, 9;
10, 11;
11, 11;

23. Close the *coll* window and, when asked to store changes, click *Save.*

The indexes in the list are labeled *0–11*. If we send the number *1* to *coll*'s inlet, it will look at index *1* and output the element at that address: *0*. If we send *coll* the number *6*, it will look at index *6* and output the number *5*. The data at address *10* is *11*, so a *10* sent to *coll*'s inlet will output the number *11* out its outlet.

Our initial objective was to filter nondiatonic pitches to the diatonic pitches of C Major. Since we have already separated incoming pitches into pitch classes and octave designations, we can use the *coll* we just made to compare incoming pitch classes to those of the desired scale. For instance, if the incoming pitch class is a *1* (C♯), the *coll* list will see the element at index *1* and output the element at index *1*: *0* (C). A *3* (D♯) sent to *coll* will output a *4* (E). In essence, we have created a *coll* that filters all incoming pitch classes to the notes of the C Major scale. Now, any chromatic notes running through *coll* will be substituted for diatonic ones.

Note that we are still dealing with pitch classes, which makes this filtering process much easier. Think about the alternative if you didn't work with pitch classes. You might try to work with an object like *select* specifying arguments for all 128 MIDI notes to trigger just the notes of the scale that you wanted. Big waste of time!

Let's get these pitch classes back to their original octave. To recombine this pitch class with the *random* pitch's original octave designation, multiply the octave designation by 12 using the * object, and add the number to your pitch class value using the+ object. Unlock your patch and

24. Create a new object called * with the argument *12*
25. Connect the first outlet of the *number* box receiving from / *12* to the first inlet of * *12*
26. Create a *number* box
27. Connect the outlet of * *12* to the newly created *number* box
28. Create a new object called +
29. Connect the first outlet of the newly created *number* box to the first inlet of the + object
30. Connect the first outlet of the *number* box receiving from *coll* to the last inlet of the + object
31. Create a *number* box
32. Connect the outlet of + to the inlet of the *number* box

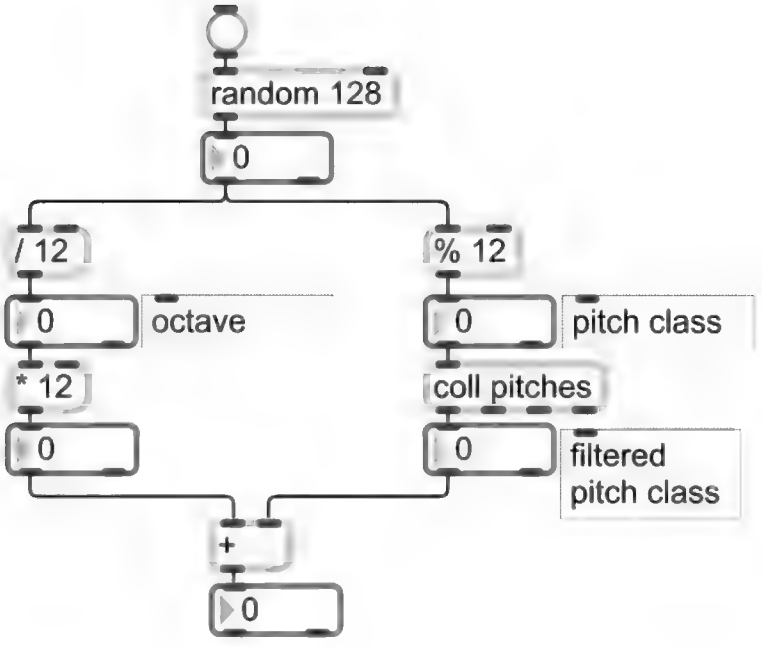


FIGURE 6.4

recombining a filtered pitch with its octave designation | random _tonal_trash.maxpat

33. Lock your patch and click the *button* connected to *random 128*

You have successfully filtered all nondiatonic pitches regardless of octave, to the diatonic notes of C Major. You can now connect the filtered note *number* box to a *makenote* and *noteout* to synthesize these pitches just like you did in the *rat _patch*. Unlock your patch and

34. Create a new object called *makenote* with the arguments *100* and *300*
35. Create a new object called *noteout*
36. Connect the two outlets of *makenote 100 300* to the first two inlets of *noteout*

Take a moment to save the file as a new file labeled *random_tonal_trash.maxpat*. Entering the pitch filter data into a *coll* is long process, especially if you intend to transpose. Luckily, *coll* has the option to "read in" premade lists as text documents. Simply send *coll* a *message* box containing the word *read*, and *coll* will open a dialog box from which you can browse for a text file. Of course, you still have to go through the process of deciding which diatonic notes the chromatic notes should be filtered to in each mode. That can be an obnoxious chore, so, for some reason, I went ahead and did that all for you.

The Modal Object Library contains the *modal_coll_filter* object, an object that receives the pitches as a list from the last outlet of *modal_change* object and automatically formats a filter list, just like we made earlier, into a *coll* object connected to its outlet.

37. Create a new object called *modal_change*
38. Create a *message* box containing the letter *C* above the *modal _change object*

39. Connect the outlet of the *message C* to the first inlet of *modal_change*
40. Create a message box containing the word *major* above the *modal_change object*
41. Connect the outlet of the *message major* to the last inlet of *modal_change*
42. Create a new object called *modal_coll_filter*
43. Connect the last outlet of *modal_change* to the first inlet of *modal_coll_filter*
44. Create a new object called *coll* with the argument *pitches*
45. Connect the outlet of *modal_coll_filter* to the inlet of the newly created *coll pitches* object

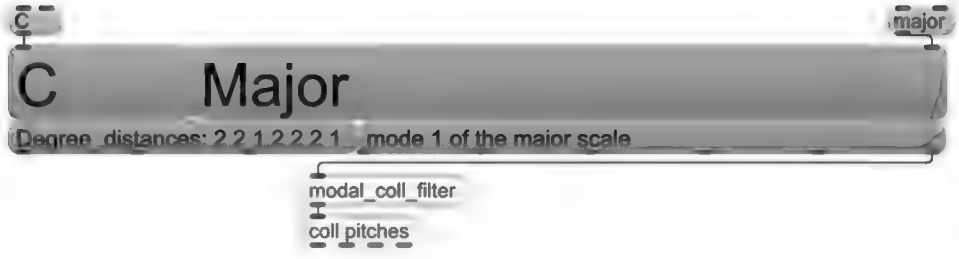


FIGURE 6.5
using modal_coll_filter to load a list into coll | excerpt of random_tonal_trash.maxpat

46. Lock your patch and click the 2 messages *C* and *major* causing the *modal_change_object* to send scale data to the *modal_coll_filter* object which will then send out index/element pairs to the *coll pitches* object

One of the best things about the *coll* object is that its argument, the name, can be used to create multiple instances of the same *coll*. Multiple *coll* objects with the same name will contain the same data even if they are in two different open patches.

In this example, the *coll* will receive the filter data from *modal_coll_filter* and store it in the *coll* object named *pitches*. Since *pitches* is the same name that we used for the *coll* object connected to the % object, the data will change in that *coll* object as well, even though they are not connected. In fact, if you had another open patch with a *coll* named *pitches*, the data would be the same in there as well.

47. Double click each *coll pitches* object to see that the data are shared between them simply because they have the same name as an argument

Abstractions and Subpatchers

Now, let's take a short aside and discuss the *modal_coll_filter* object, which isn't really an external object in the sense we've been talking about them, but is

actually just a Max patch that we're using like an object. This is called an "abstraction." Unlock your patch and

48. Ctrl+click (Mac) or right click (Windows) the *modal_coll_filter* and go down to *Object* in the contextual menu and click the option *Open Original "modal_coll_filter"*

The *modal_coll_filter* abstraction is a Max patch located somewhere in the Max path. In fact, this file is bundled with the other objects you installed earlier. As long as you type in the name of the patch, it will load like an object. With the patch open, you can unlock it, and maximize the patch window to see that it contains Max objects and a few other abstractions.

Do you see the *p launchURL* object? This is also not actually an object at all but a *subpatcher*: an encapsulated patch useful for keeping the patches organized and modular; it's a patch within a patch. Using a subpatcher makes reusing parts of your patch easier.

49. Highlight the *p launchURL* subpatcher and choose *Edit>De-encapsulate* from the top menu

De-encapsulating a subpatcher expands the contents of the subpatcher in the main patch.

50. Undo this action by selecting *Edit>undo* from the top menu

Note the small brown object directly beneath the subpatcher. This is an object called *inlet*. It allows data to be sent into this abstraction when the abstraction is loaded in another patch. Since there is one *inlet* in this abstraction, when a user creates the object *modal_coll_filter*, it will have one inlet.

Find the subpatcher called *p major* in the middle of the patch. With the patch still unlocked

51. Hold the ⌘ (Mac) or ctrl (Windows) key down and double click the *p major* subpatcher to display the subpatcher's contents in a new window[3]

A window containing the subpatcher's contents will open. Notice that this subpatcher also contains the brown *inlet* object we just saw along with several *match* objects. This subpatcher receives whole step and half step patterns and matches them against a series of *match* objects.

If the subpatcher finds a match—for example, *2 2 1 2 2 2 1*, the major scale—it knows that the scale is a major scale and can trigger a certain set of index/element pairs to be sent to the *coll* object. The particulars of everything happening in this subpatcher are a bit involved for a discussion at this point, but

3. Note that when you hold the ⌘ (Mac) or ctrl (Windows) key down while clicking in an unlocked patch, the subpatcher window will open as it would if the main patch were locked.

our examination is useful for us to see the underpinnings of a working abstraction. Close the subpatcher window.

One benefit of creating an abstraction like this is that it's modular; every time I want to use a *coll* in this way, I don't need to copy and paste all of these necessary objects into my patch, or rewrite them. I can simply enter the name of this patch *modal_coll_filter* in an object box and I'm ready to go.

If you design your patches to be modular like this, you can reuse parts of old patches in your new ones, saving you some time. If you use Max a lot, it can save you years. Think about it: how many times do you really need to rewrite the same "*makenote* connected to *noteout*" bit? Close the abstraction without saving the changes.

There is another shortcut for pasting preassembled object configurations into your patch by ctrl+clicking (Mac) or right clicking (Windows) a blank portion of the patch and selecting "Paste From." For example, the option *playnote* inserts a *makenote* object already connected to a *noteout* object. Customized shortcuts can be added to this menu by copying patches to the *patches>clippings* folder inside the Max application folder.

Working with Pitch Classes

We can restrict the range of random notes being played to one octave by sending a *message* box with the argument *12* to *random*'s second inlet. This will restrict the range of random numbers to 0–11, the pitch classes. If you want to include the first note of the next octave, use the argument *13* instead to generate random numbers between 0 and 12. With your patch unlocked

52. Create a *message* box containing the number *12*
53. Connect the outlet of the *message* box to the last inlet of *random 128*

Since we're working with pitch classes, we will use the + object between the *random* object and the % and / objects to transpose the pitch classes to a desired octave. For example, if you want the random notes in the fifth octave, the argument for + would be *60*.

54. Disconnect the outlet of *random 128* from the inlet of the *number* box directly below it
55. Create a new *number* box
56. Connect the outlet of *random 128* to the inlet of the newly created *number* box (Note: you may have to reposition the *random* object higher in your patch to make more room)
57. Create a new object called + with the argument *0*
58. Connect the first outlet of the newly created *number* box (Step 55) to the first inlet of + *0*

59. Create a *message* box containing the number *60* directly next to the *message 12*
60. Connect the outlet of *message 60* to the last inlet of *+ 0*
61. Create a *button* above both *message* boxes
62. Connect the outlet of the *button* to the first inlet of both *message 12* and *message 60*
63. Create a *message* box containing the number *128* to the far right of the *message 60*
64. Connect the outlet of *message 128* to the second inlet of *random 128*
65. Create a *message* box containing the number *0* to the right of the *message 128*
66. Connect the outlet of *message 0* to the second inlet of *+ 0*

We now have two modes of operation for the generation of random pitches: "full range" mode, which spans all *128* MIDI notes, and now, "one octave" mode which spans just a single octave. We can use *button*s as controls to switch between these two modes.

67. Create a *button* above both of these newly created *message* boxes
68. Connect the outlet of the *button* to the first inlet of both *message 128* and *message 0*

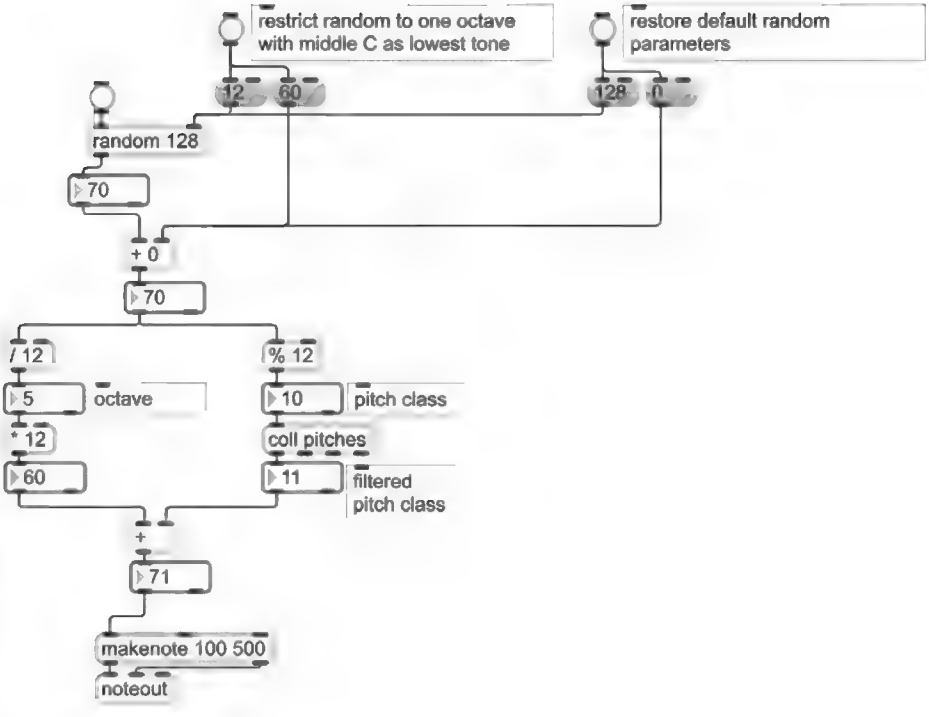


FIGURE 6.6

trigger arguments for random tonal pitches in one octave | random _tonal_trash.maxpat

69. Lock your patch and click on the *button* connected to *12* and *60*, then click on the *button* connected to *random*. This restricts the outputted random numbers to a one-octave chromatic scale where the lowest pitch will be middle C (pitch class *0* + *60*)
70. Click the *button* connected to *128* and *0*, then click on the *button* connected to *random*. This restores the default values for *random*, returning the *random* output to *128* numbers without a transposition

Because of the arguments we supplied to *random* and +, *128* and *0* will be the values the patch uses by default when it loads.

Alternate Way

Let's look at an alternative way of doing the same thing by using the *expr* object to write out much of the process we just did as a single mathematical expression. This may seem intimidating at first, but it really isn't. Just stay with me.

Find some blank space in your patch. Unlock the patch and

71. Create a new object called *random* with the argument *12*
72. Create a *button*
73. Connect the outlet of the *button* to the first inlet of *random 12*

That much is review. Let's keep going:

74. Create an object called *expr* with the following expression[4] as an argument: *$i1 + ($i2*12)*
75. Create 3 *number* boxes
76. Connect 2 *number* boxes to the 2 inlets of *expr $i1 + ($i2*12)*, respectively
77. Connect the third *number* box to the outlet of *expr $i1 + ($i2*12)*

In Chapter 4, we talked about the $ as a placeholder; *expr* uses it in a similar way. In our *expr* argument, we've used the placeholder *$i1*, in which the dollar sign is a placeholder for incoming data, the *i* specifies that the incoming datum is an integer (as opposed to a floating-point number), and the *1* means that the data are coming in through the object's first inlet.

4. Note that for *expr,* the spaces used between arguments in the expression do not matter. Expressions can be written as *$i1+($i2*12)* or *$i1 + ($i2 * 12)* to achieve the same result.

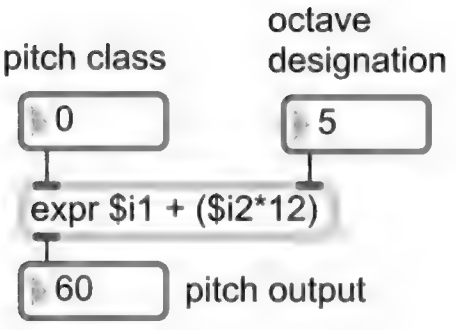


FIGURE 6.7
formatting expr | subpatcher in random _tonal_trash.maxpat

In this expression, the number in the first inlet of *expr* ($i1) will be added (+) to the product of the number received in the second inlet ($i2) and 12. Altogether, the expression reads *$i1+($i2*12)*.

In a moment, we're going to connect the outlet of *random 12* to the first inlet of *expr*. We know that the value coming into *expr*'s first inlet is going to be an integer between 0 and 11 which we want to transpose up some number of octaves just as we did before. Using *expr,* the number of octaves to transpose will be determined by the number sent to *expr*'s second inlet.

In other words, the pitch class in the first inlet of *expr* will be added to the product of 12 multiplied by an octave number. We used 12 in this expression because that's the number of pitches in an octave. We multiplied 12 by 5 because we want our pitch classes transposed up 5 octaves. Since 12 times 5 equals 60, every pitch class received in the first inlet will have 60 added to it which is the same as transposing a note up 5 octaves.

The desired octave number is written in *expr* using *$i2* as a placeholder for an incoming value. Because of the *2* in the *$i2*, this value will be received in *expr*'s second inlet. The great thing about this octave implementation with *expr* is that we can easily change the number of transposed octaves by changing the number sent to *expr*'s rightmost inlet.

For this mathematical expression to be executed in the proper order, we've enclosed the multiplication portion of the expression arguments in parentheses since we need the value received in the second inlet to be multiplied by 12 before we add it to the value received in the first inlet. Do you remember using parentheses in middle school math when discussing "order of operations"? You've finally answered that question you asked way back then: "When am I ever going to use this in real life?" Math teachers the world over, rejoice!

78. Create a new object called *coll* with the argument *pitches* (Remember: this *coll* will have the same contents as the other *coll* in your patch with the same name)
79. Connect the outlet of *random 12* to the inlet of *coll pitches*
80. Connect the first outlet of *coll pitches* to the *number* box connected to *expr*'s first inlet

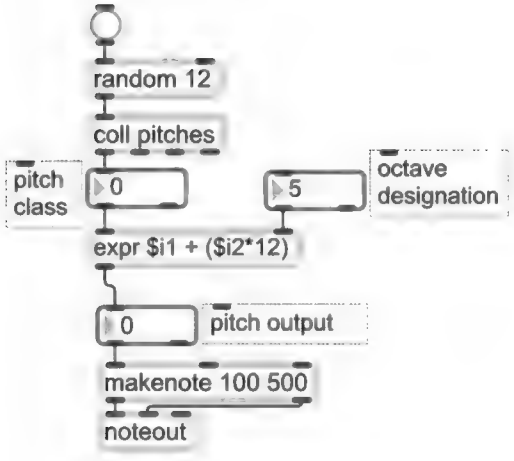


FIGURE 6.8
subpatcher showing the same action using expr | random_tonal_trash .maxpat

81. Create a new object called *makenote* with the arguments *100* and *500*
82. Connect the first outlet of the *number* box receiving from *expr*'s outlet to the first inlet of *makenote 100 500*
83. Create a new object called *noteout*
84. Connect both outlets of *makenote 100 500* to the first 2 inlets of *noteout,* respectively

When you enter a number in the *number* box connected to *expr*'s second inlet—for example, the number 5—all pitch classes received in *expr*'s first inlet will be transposed up 5 octaves. We could have also substituted the $i2 in our expression for the number 5. However, by not doing so, we give the patch the flexibility to change the octave designation, which it could not do if we had "hard coded" the number 5 into the expression.

85. Create a new object call *loadmess* with the argument *5*
86. Connect the outlet of *loadmess 5* to the inlet of the *number* box connected to the second inlet of *expr*

By adding the *loadmess 5* to the patch, the number *5* will be sent to the *expr* object's second inlet when the patch loads up, yet we'll still have the freedom to change the octave designation via the *number* box without having to unlock the patch.

87. Highlight all the objects you've created in Steps 71–86 and select *Edit>Encapsulate* from the top menu
88. Double click in the newly created subpatcher and, next to the *p*, enter the text *using_expr* with a space after the *p*. This gives a somewhat descriptive name for our subpatcher

Congrats! We created an alternate way of doing what we did before. Regardless of which method you choose to get the result you want, the important thing is that you *get* the result you want. In Max, there are numerous programming

approaches that you can take to reach the same conclusion. Some approaches, like the *expr* approach taken here, may involve fewer objects than others, which could help conserve your computer's processing power and system resources. As you continue to learn Max, you may find yourself doing things with only a few objects that you previously did with many objects.

When you reopen this patch, you will have to manually click on the *message* boxes *C* and *major* connected to *modal_change* before you can filter random chromatic notes to diatonic ones. Let's implement some *loadbang* objects so the patch automatically performs this task for you:

89. Create a new object called *loadbang*
90. Connect the outlet of the *loadbang* object to the first inlet of the *message C* and the *message major* boxes

When the patcher loads, these *message* boxes will receive a bang from the *loadbang* object. Note that we could have also used the *loadmess* object with the argument *C* (or *major*) to get the same result. Save and close your patch.

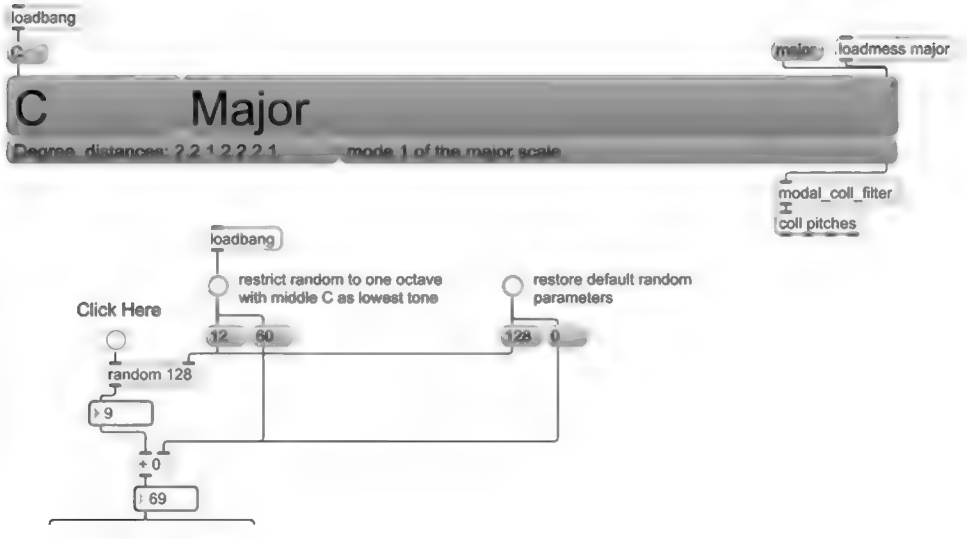


FIGURE 6.9

loadbang and loadmess trigger actions on load | excerpt random_tonal _trash.maxpat

The ability to filter all notes to a single 7-note collection could allow you to design musical instruments for certain populations such as special needs individuals and those without musical training that allow them to create tonal music with ease and flexibility. In the next chapter, we will discuss the creation of accessible musical instruments and controls for composition and performance.

Tables

Another way to store data in Max is in a *table* object. Begin a new patch, and

1. Create a new object called *table*
2. Create 2 *number* boxes: one above the *table* and another below the *table*
3. Connect the first outlet of the *number* box above the *table* to the first inlet of the *table*
4. Connect the first outlet of the *table* to the first inlet of *number* box beneath the *table*

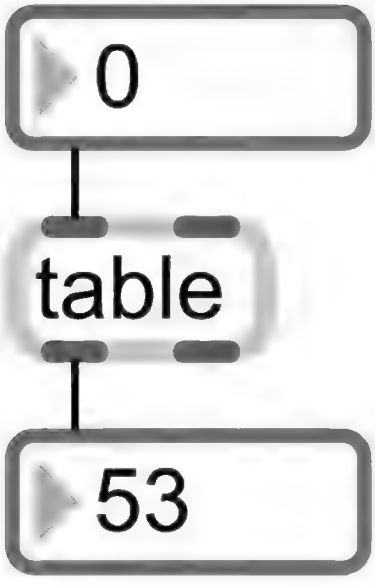


FIGURE 6.10
a number in table's inlet gets data at that address | table_trash.maxpat

5. Lock the patch and double click the *table* object to open a window displaying a graphical representation of the data stored in the *table*; currently, there are no data present

The way data are stored in a *table* is similar to the way they are stored in a *coll*. One main difference is that *coll* can work with data other than numbers while *table* can work only with numbers. However, *table* is a graphical object and could be better suited for applications that involve user interaction or visual representation of the numbers in the table.

Like *coll*, *table* elements are stored at an index. In this case, *table* stores numerical elements according to its vertical (Y) value at the index of its horizontal (X) value.

6. Move your mouse over the *table* until the coordinates "*x 0 y 53*" appear on the screen, and then click on that point in the *table*
7. Move your mouse slightly to the right and click when you are over the coordinates "*x 1 y 46*"



FIGURE 6.11

a table object reflecting two coordinates at the far left | table_trash.maxpat

8. Close the *table* window and type *0* into the *number* box connected to *table*'s inlet. You will need to press the *return* or *enter* key to actually send the number or you may simply click elsewhere in the patch

Notice that the number *53* is received in the *number* box connected to *table*'s outlet; this is the element stored at the index *0*. A user could store a collection of notes from a scale in a *table* where each index value contained a different scale degree as its element. Since *table* is graphical, it's also feasible for a user to draw a collection of notes right into the object.

9. Open up the *table* again and draw a line from one corner of the *table* to another

Unlock your patch and

10. Create a *button*
11. Connect the outlet of the *button* to the first inlet of *table*

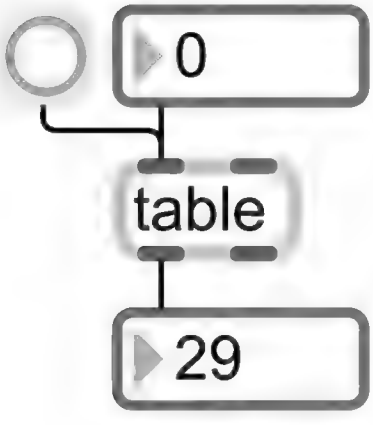


FIGURE 6.12

a button randomly gets data value from the table| table_trash.maxpat

12. Lock your patch and click the *button* to send an element (*Y*) from an index (*X*) in the *table* to its outlet. It will appear that each bang sends out a random number from the table

The element being sent is not chosen randomly, but is instead a weighted selection according to the value of the element on the *Y* axis. For instance, in an earlier example, the element (*Y*) was *53* at index (*X*) *0*, and the element was *46* at index *1*. Since these were the only two data points in our table, the element at every other index was at the default value of *0*.

When a bang is sent to the *table*, instead of outputting an element value, *table* outputs an index value according to the value of each element. Since the element *53* is a higher number than *46*, there is a slightly greater probability that a bang sent to *table* would output index *0*, the index containing element *53*, than index *1*, the index containing *46*. If we were to lower the element of index *0* to, for example, *5*, then there would be a greater probability that *1* would be outputted when a bang was sent to *table* since *1* is the index of the element *46*. If all of the index values in your *table* are at an equal element value (height), the results from a bang will be evenly distributed.

13. Open the *table* and enter the following values by clicking on the corresponding data points (You may want to enlarge the *table* window in order to see the data points):

x 0 y 1
x 2 y 1
x 4 y 1
x 5 y 1
x 7 y 1
x 9 y 1
x 11 y 1
x 12 y 1



FIGURE 6.13
C Major pitch class values evenly distributed in a table | table_trash.maxpat

The indexes you entered into the *table* (*0 2 4 5 7 9 11 12*) are the pitch classes[5] from the C Major scale. Since all of the indexes (pitches) are at the same height (element value), a bang sent to the *table* will randomly choose one of the indexes, which are pitch classes, since they all have the same element value, *1*.

5. Plus the octave: *12*

The element value *1* is the weighting for the random selection of each pitch class. Since the indexes at *1 3 6 8 10*, the chromatic pitch classes, have an element value of *0*, they will never be chosen when a bang is sent to the table. A user could then send a bang to the *table* and output only diatonic pitch classes for the key of C Major without the need to filter the data to the appropriate key as we did in the *coll* example.

14. Click the *button* to output diatonic pitch classes from the *table*

Just as the *modal_coll_filter* object formatted the selected scale into a *coll* filter list, a similar abstraction in the Modal Object Library called the *modal_table_filter* formats the scale data into a *table* when used with *modal_change*. Unlock the patch and

15. Create a new object called *modal_change*
16. Create a new object called *modal_table_filter*
17. Connect the last outlet of *modal_change* to the first inlet of *modal_table_filter*
18. Delete the *number* box and *button* from the inlet of *table* as we no longer need these
19. Connect both outlets of *modal_table_filter* to the 2 inlets of the *table* object
20. Create a new object called *loadmess* with the argument *C* above *modal_change*
21. Connect the outlet of *loadmess C* to the first inlet of *modal_change*
22. Create a new object called *loadmess* with the argument *major* above *modal_change*
23. Connect the outlet of *loadmess major* to the last inlet of *modal_change*
24. Lock your patch and double click both *loadmess* objects causing their arguments to be sent to *modal_change*

Like *coll*, *table* objects can also be named with an argument so that all *table* objects sharing the same name will share the same data. Unlock the patch and

25. Double click the *table* object and give it the argument *my_table*



FIGURE 6.14

two table objects sharing the same name receive pitch class data| table _trash.maxpat

We can now reuse parts from previous patches to automatically generate random tonal pitch classes, move these pitch classes to a desired octave, and synthesize them. On a blank part of your unlocked patch

26. Create a *toggle*
27. Create a new object called *metro* with the argument *500*
28. Connect the outlet of *toggle* to the inlet of *metro 500*
29. Create a *button*
30. Connect the outlet of *metro 500* to the inlet of the *button*
31. Create a new object called *table* with the argument *my_table*
32. Connect the outlet of the *button* to the first inlet of *table my_table*
33. Create a *number* box
34. Connect the first outlet of *table my_table* to the inlet of the *number* box
35. Create a new object called + with the argument *60*
36. Connect the first outlet of the *number* box to the first inlet of *+ 60*
37. Create a new object called *makenote* with the arguments *100* and *500*
38. Connect the outlet of *+ 60* to the first inlet of *makenote 100 500*
39. Create a new object called *noteout*
40. Connect the outlets of *makenote 100 500* to the first 2 inlets of *noteout*

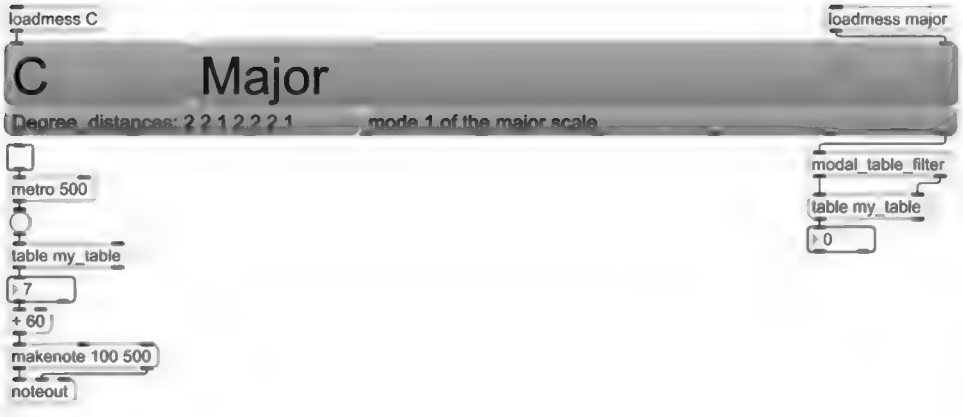


FIGURE 6.15

create a random tonal table with probability weightings | table_trash .maxpat

41. Lock the patch and turn on the *toggle* to make the *table* randomly generate the pitches of the C Major scale. If you change the scale type through *modal_change*, the pitch class data in the *table* will automatically change as well

The patch looks great, but what about something a little more visual? One of the coolest parts of the *table* is the graphical display, but it's hidden until you

double click the *table* object. The *itable* object, however, displays the *table* data as a window within the patch. Unlock your patch and

42. Double click the *table my_table* object you created earlier, and replace all text in the object with the word *itable*
43. Highlight and move the objects that *itable* is now on top of so that they are beneath the *itable*

For the *itable* to share the data of *table my_table,* we need to give it the name *my_table*. This is done through the *Inspector*.

44. Open the *Inspector* for *itable* and change the *Table Name* to *my_table*

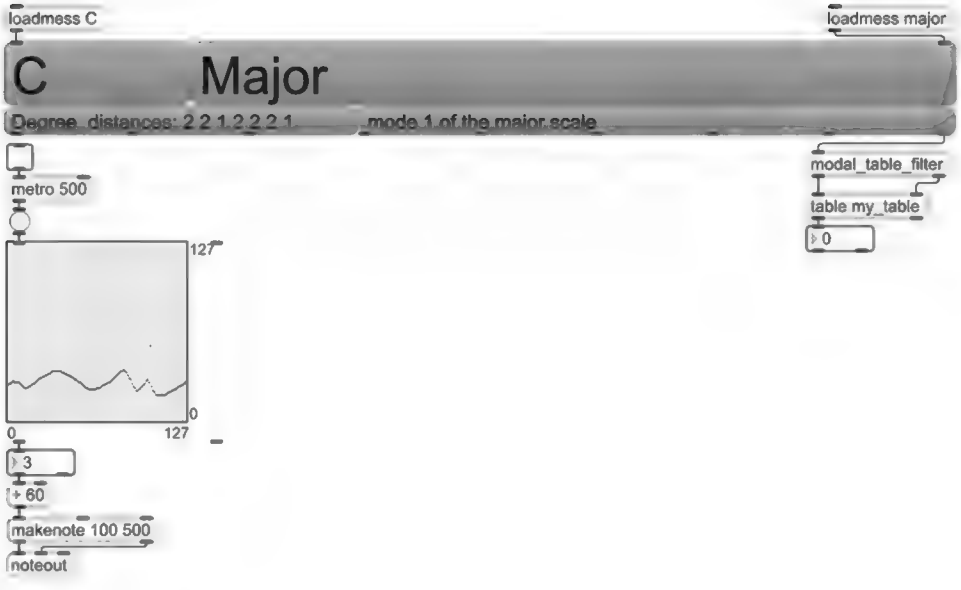


FIGURE 6.16

itable used as inline graphical table | table _trash.maxpat

45. Lock the patch and turn on the *toggle* to ensure that *itable* is working as well as the *table* did

Great job! Loading pitch collections into a *table* is a great way to store scales and chords for performance use. If you decide that you'd rather keep your patch atonal and draw your own pitches into *itable*, you will probably want to get rid of the *+ 60* object we used to transpose the pitch classes or else your MIDI pitches will be way out of range. Save this patch as *table_trash.maxpat.*

Remember, the objects and abstractions from the Modal Object Library that I've shown you in this chapter are only shortcuts for doing things we've already discussed. In future chapters, we will be using these and other objects and abstractions in our discussion to speed things along. Feel free to use them as much as you'd like or not at all. Everything we've been discussing can be done without the use of these third party objects.

New Objects Learned:

- *modal_change* [external object]
- *%*
- */*
- *coll*
- *modal_coll_filter* [abstraction]
- *inlet*
- *match*
- *expr*
- *table*
- *modal_table_filter* [abstraction]
- *itable*

Remember:

- A collection of data is typically stored in the following format: *index number*, *stored data* (called an *element*).
- If possible, perform filtering calculations on pitch classes rather than on the entire range of possible pitches.
- Paste preassembled object configurations into your patch by ctrl+clicking (Mac) or right clicking (Windows) a blank portion of the patch and selecting "Paste From"

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Data Structures and Probability—Working with histograms and lookup tables
  - Data Collections—Using databases
  - Encapsulation—Patchers inside of patchers
  - Abstractions—Creating libraries of reusable code

External Objects Database Concise Reference List:

- Cycling '74 "Projects" and "Toolbox"—http://www.cycling74.com
- Max Objects Database— http://www.maxobjects.com
- CNMAT's MMJ Depot—http://cnmat.berkeley.edu/downloads
- Jamoma—http://www.jamoma.org/

- ppooll—http://ppooll.klingt.org/
- UBC Toolbox—http://debussy.music.ubc.ca/muset/toolbox.html

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows you to perform, in some way, strictly in a diatonic key. Give your patch the ability to easily change keys. Use your knowledge of UI (user interface) objects like *slider* in addition to MIDI input objects as you devise your instrument.
- Create a patch that takes ASCII keyboard numbers and uses them as diatonic pitches for some tonic and mode.

7

Control Interfaces

In this chapter, we will examine some premade patches demonstrating a few techniques for designing diatonic musical instruments. We will review some of the basic ins and outs of MIDI, learn some ways to program more efficiently, and discuss a number of control options for your patches. As we examine some working patches and encounter many new objects, please read carefully and, in your mind, follow the flow of data from one object to the next as the process is described in the text. Remember that it will be beneficial to look at the Help file for any objects you may have forgotten about or do not fully understand as you encounter them in this chapter.

In the last chapter, you installed the *EAMIR SDK* (*Software Development Kit)* which, in addition to putting the Modal Object Library into the Max search path, put a bunch of patches I've created into the path as well. In fact, if you select *Extras* from the top menu, you will see an item marked *EAMIR* among the other extras.[1]

1. Click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
2. In the *umenu* labeled *Examples*, click the first item *1.EAMIR _MIDI_Basics.maxpat*

1. If you did a manual install of the EAMIR SDK, you will need to locate the *EAMIR_SDK* folder on your computer and open the file *EAMIR.maxpat*.

bpatchers

A patch will open containing two rectangular boxes. Unlock the patch to see that these two rectangular boxes have inlets and outlets just like the objects we've been working with, except that they, in themselves, contain other objects. These two boxes are called *bpatcher*s.

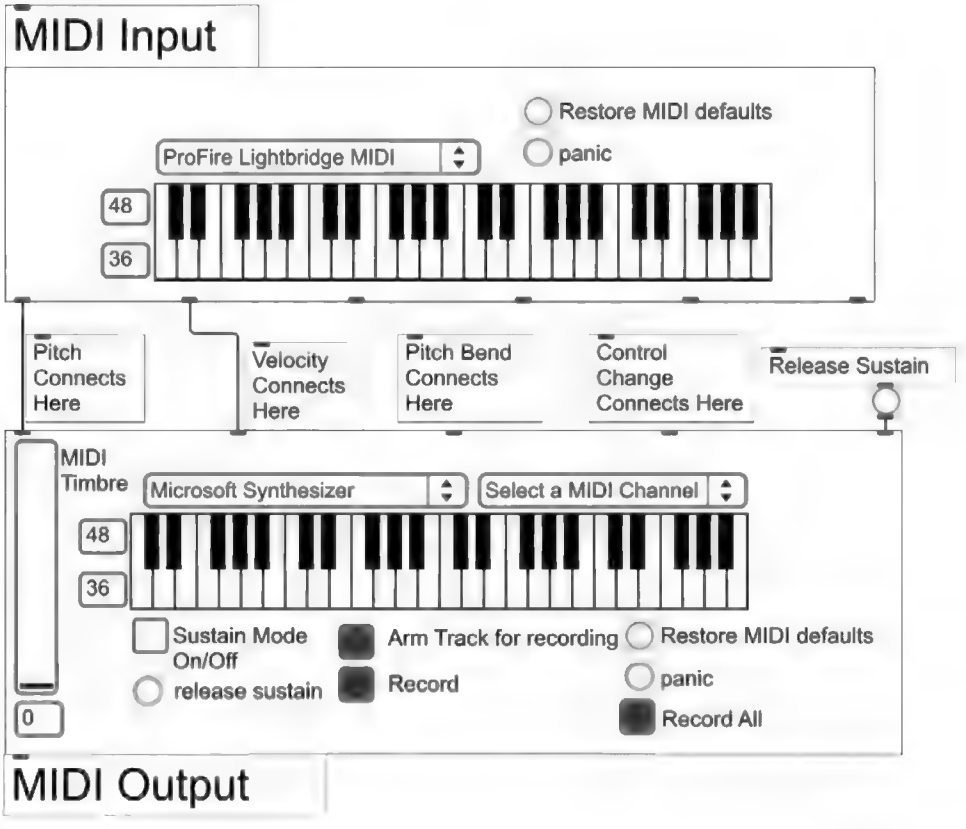


FIGURE 7.1

two bpatchers showing a number of MIDI-related objects | EAMIR_MIDI _Basics.maxpat

A *bpatcher* is an object that allows an existing Max patch to be loaded into a viewable window; that window is the *bpatcher* object itself.

3. Open the *Inspector* for the upper *bpatcher*

Note that the line *Patcher File* within the *Inspector* displays the filename of the Max patch currently loaded in the *bpatcher*: *EAMIR_MIDI_in.maxpat*. This is just a Max patch that is being displayed in the *bpatcher*. If you were to select the *Choose* button next to *Patcher File*, you would be able to load up any Max file into that *bpatcher*.

4. Close the *Inspector*
5. Ctrl+click (Mac) or right click (Windows) the upper *bpatcher* and select *Object>Open Original "EAMIR_MIDI_in.maxpat"* from the contextual menu

Selecting "Open Original" opens the actual patch that is loaded in the *bpatcher*. Any changes you make to this patch will also change in the *bpatcher* referring to the file.

This patch is set to open in Presentation view. Unlock the patch and put it in Patching mode.

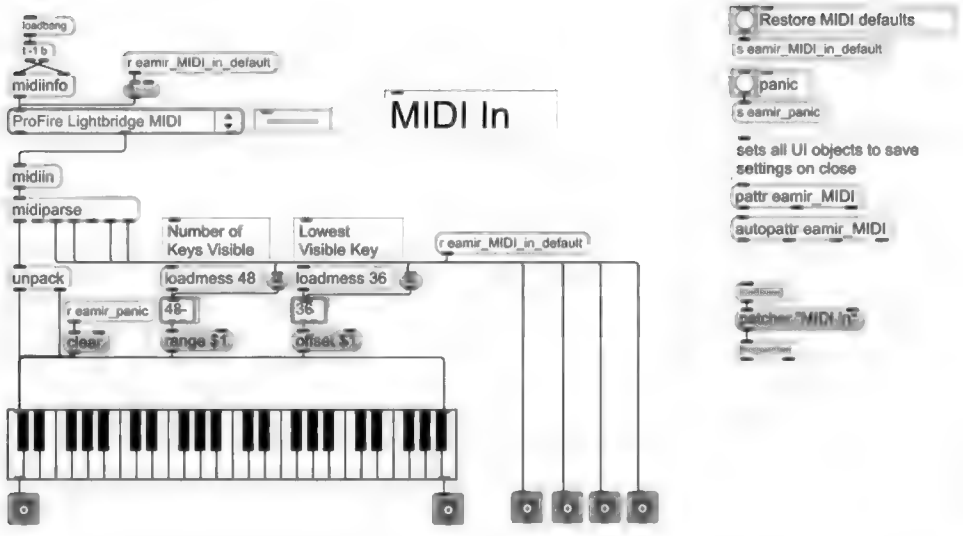


FIGURE 7.2

MIDI-related objects within a bpatcher | EAMIR_MIDI_in.maxpat

The contents of this patch aren't that impressive. There are just some basic objects you'll need to get MIDI into your patch. One difference is that instead of using *notein*, I used the object *midiin* to receive MIDI data from my MIDI device and send out raw MIDI data (not just notes as with the *notein* object) to *midiparse*. The *midiparse* object separates the raw MIDI data into different types: notes, bend, control, channel, and so on.

The first outlet of *midiparse* outputs pitch and velocity pairs as a list. By looking in the patch, you can trace the path of the first outlet of *midiparse* to an *unpack* which separates the pitch and velocity values. The pitch values are sent from *unpack*'s first outlet and velocity values are sent from its second outlet. Both the pitch and velocity values are then sent to a *kslider* which is provided here simply for visualization purposes. Each of *kslider*'s outlets is connected to an object called *outlet*.

The *outlet* object is a way to get data out of an embedded patch. Just as in the abstraction and the subpatcher we looked at in the last chapter, *bpatcher* objects generally rely on *inlet* and *outlet* objects to get data to and from their inner contents. When this patch is embedded into a *bpatcher*, it will show 6 outlets in the main patch because there are 6 *outlet* objects in the embedded patch. If we added an additional *outlet* object to the embedded patch, it would show 7 outlets in the main patch. Data sent from the first outlet on the far left of the embedded patch will be the data sent through the first *outlet* object in the main patch.

6. Open the *Inspector* for the *outlet* receiving from *kslider*'s first outlet

Notice that I have added the word *pitch* next to the *Comment* line in the Inspector. When this patch is loaded into a *bpatcher*, you can hold your mouse over the first outlet of the *bpatcher* to see the word *pitch*, revealing that pitches come out of this first outlet. Each of the *outlet* objects in this patch have a comment describing the type of data output. Adding a comment allows you to remember what outlet certain data is coming from.

The topmost portion of the patch is just a fancy way of displaying the MIDI devices connected to your computer. The *midiinfo* object reports a list of your computer's currently available MIDI output devices when it receives a bang in its first inlet. A *-1* sent to *midiinfo*'s second inlet causes it to output your computer's MIDI input devices.[2]

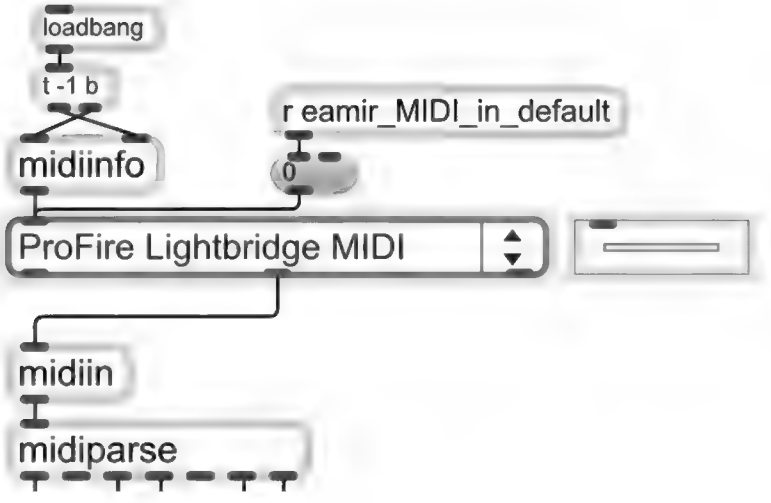


FIGURE 7.3

populating a list of MIDI input devices into a umenu (devices listed in umenu will vary) | EAMIR _MIDI_in.maxpat

The *loadbang* sends a bang to the *t*, or *trigger,* object which first sends a bang to *midiinfo*'s first inlet and, second outputs a *-1* to the second inlet of *midiinfo*. The list of available input devices is sent from *midiinfo* and populated in a *umenu*. When a user selects one of these MIDI devices from the *umenu*, the *midiin* will receive raw MIDI data from that device. This is useful if you have more than one MIDI device connected to your computer.

The small rectangle to the right of the *umenu* is an object called *hint*.

7. Open the Inspector for *hint*

The *hint* object was given the *Hint* "MIDI Input Device." In Presentation mode, the *hint* object can be layered on top of the *umenu* so that if a user holds the mouse over the *umenu*, the phrase "MIDI Input Device" will appear. Informa-

2. A *1* can be sent to the second inlet to, once again, cause *midiinfo* to output your computer's MIDI output devices.

tion like this helps to make your patch more accessible. There are several *hint* objects in this patch. Note that all objects have the option to set a hint accessible through the *Inspector*.

Look at the 3 *message* boxes above the *kslider*. We've already discussed, in Chapter 4, how the *clear* message will deselect any highlighted keys on the *kslider*, but what about the *range $1* and *offset $1* messages? Where did they come from?

8. Open the *Inspector* for *kslider*
9. In the *Inspector*, click on the phrase "Low MIDI Key Offset" and drag the phrase directly into your patch

Notice that the phrase becomes a *message* box formatted to send a message that *kslider* understands. We remember that the *$1* part is placeholder for some data. As you can see in the patch, a *number* box is sent to the first inlet of the *messageoffset $1* so that the lowest visible key in the *kslider* can be changed on the fly by substituting the value in the *number* box. For most objects, messages can be dragged from the *Inspector* and used to change properties for the object in this way.

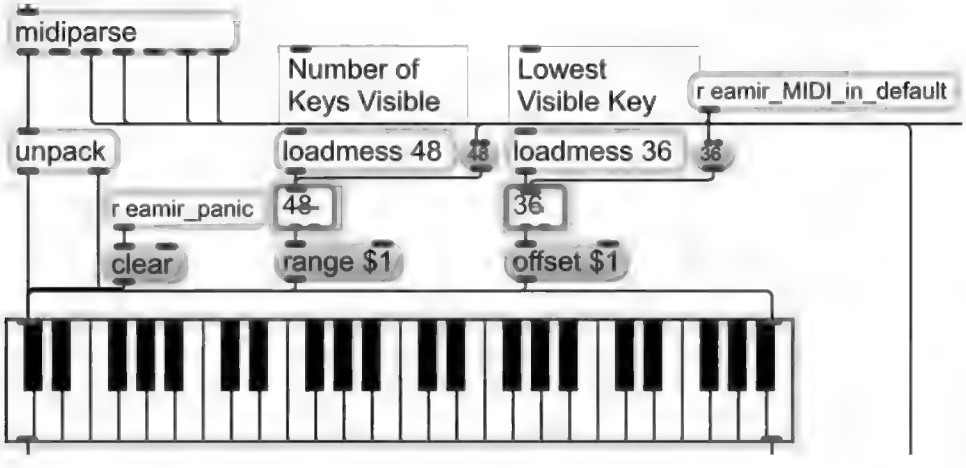


FIGURE 7.4

changing visual attributes of kslider without going to the Inspector | EAMIR _MIDI_in.maxpat

You must be wondering what the "*r eamir_MIDI_in_default*" is all about. Well, the object *r* is short for the object *receive* which is used in combination with the object *s*, or *send*, to transmit data wirelessly (or "patchcord-lessly").

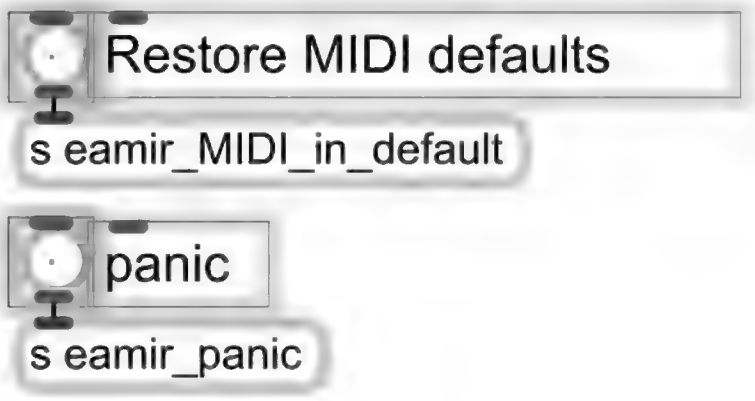


FIGURE 7.5

send objects (abbreviated "s") transmit data to receive objects (abbreviated "r") with the same name | EAMIR_MIDI_in .maxpat

The argument I've specified for the *r* object, in this case, is the name *eamir_MIDI_in_default*. I've created an *s* object with the same name. Anything sent to the *s* will be received by the *r* with the same name regardless of where they are located on the screen or even if they are in two different open patches. I've attached a *button* to the inlet of *s eamir_MIDI_in_default* so that when it is pressed, a bang will be sent to *r eamir_MIDI_in_default* which will trigger the *message* boxes containing *48* and *36* to change the *range* and *offset* of the *kslider*. Now, if someone changes the appearance of *kslider*, I'd need only to click the *button* to restore the default values. It's a nice way to foolproof your patch. Note that the *umenu* at the top is also receiving wirelessly from the *s* object.

10. Lock the patch and increase the number in the *number* box above the *message range $1* thus changing the number of visible keys in the *kslider*
11. Click the *button* connected to *s eamir_MIDI_in_default* to restore the size of the *kslider*

We'll discuss the *pattr* and *autopattr* objects in a moment. For now, close this patch without saving anything. If you absolutely love something you just did in the patch, save a copy of it (*File>Save As*) because anything that changes in this patch will change in all *bpatcher* instances in the rest of the EAMIR SDK (and there are plenty of instances of this). Of course, the EAMIR SDK is free and open-source so you can modify and improve on it any way you'd like. Let's look at that other *bpatcher*.

1. Ctrl+click (Mac) or right click (Windows) the lower *bpatcher* and select *Object>Open Original "EAMIR_MIDI_out.maxpat"* from the contextual menu

This patch is also set to open in Presentation mode. Unlock the patch and put it in Patching mode.

MIDI Out bpatcher

This patch is constructed similarly to the patch we just looked at. There are a few things in particular that we should discuss. Of the most important is the presence of *midiout* in place of *noteout* just as the "MIDI In" *bpatcher* used *midiin* in place of *notein*. In this patch, each element of the MIDI message is formatted into a single message using the *midiformat* object. Just as we used *unpack* to separate pitch and velocity messages coming from *midiparse*, we use *pack* to put the *pitch* and *velocity* messages back together for *midiformat*.

A few other objects including *sustain* and *flush* have been added to this patch as well. We've already discussed *flush*, in Chapter 4, and one application for *flush* which involved making our generated chords legato. Since *flush* handles note-off messages, it also works well as an all around "panic" button in case of a stuck MIDI note.

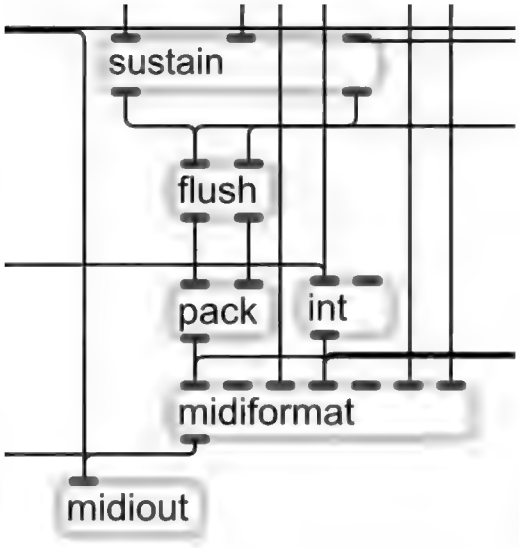


FIGURE 7.6
formatting a MIDI message | EAMIR_MIDI_out.maxpat

The *sustain* object functions similarly to the sustain pedal on a piano but uses a *toggle* instead of an actual pedal. When the *toggle* is on, the notes will sustain until the *toggle* is turned off—a useful thing to have in a MIDI out patch even if you don't use it all the time.

For now, we will have to resist the urge to discuss the MIDI recording functions of this patch until Chapter 10. Suffice it to say that this patch has MIDI recording capabilities.

Saving Settings

Here's a predicament: you get your patch set up by selecting the correct MIDI input device from the *umenu,* you choose the timbre you like from the *slider*, and you get your *kslider* to the exact range and offset you like. The problem is that when you close your patch, all of those adjustments you made are lost and have to be redone when you open the patch again. Imagine using a patch like this in a classroom, performance, or research project when you have to go through a bunch of steps just to get started: major inconvenience! Thankfully, the *pattr* object can easily fix all of your problems (or, at least this trivial problem) by saving information about your patch's settings.

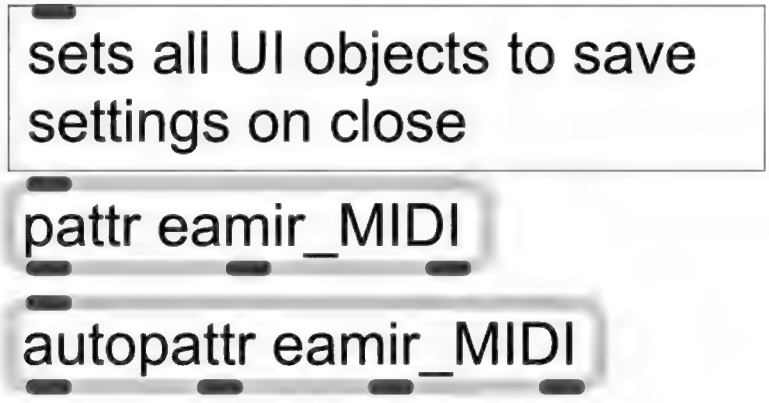


FIGURE 7.7
storing presets with pattr | EAMIR_MIDI_out.maxpat

The *pattr* object takes a name as its argument and saves the settings from objects with the same name. Now, you can't very well add the name you gave to *pattr* at the end of a UI (user interface) object like *slider* because the object won't understand it. How do you name objects to be used with *pattr*? Simple! Open the *Inspector* for an object and give its *Scripting Name* the same name you gave to *pattr*. The *autopattr* object takes the same name and is used to allow multiple objects in the patch to have their data stored in *pattr*.

2. Open the *Inspector* for *slider*

Notice that *slider*'s *Scripting Name* is *eamir_MIDI[4]*. The *pattr* storage system works in connection with *autopattr* to assign a number, in the case of *slider*, [4], to the *Scripting Name* so that multiple objects can have their settings stored in *pattr*. All you have to type in to the *Scripting Name* field is the name you gave to *pattr*.

3. Lock the patch and change the *slider* value to some number of your choosing
4. Save and close the patch (really, it's OK)
5. Open the same patch once again by ctrl+clicking (Mac) or right clicking (Windows) the lower *bpatcher* and select *Object>Open Original "EAMIR_MIDI_out.maxpat"* from the contextual menu
6. Notice that the *slider* value is exactly where you left it
7. Change the *slider* value back to *0* and save the patch
8. Close both the patch *EAMIR_MIDI_out.maxpat* and the patch *EAMIR_MIDI_Basics.maxpat*

Creating Tonal Adaptive Instruments

My background is not as a programmer, but as a musician. Before I taught at the university level, I taught K–12 music during the day and attended grad school classes at night. As a beginning teacher, one difficulty I en-

countered in my classroom was my lack of anticipation regarding the diverse levels of musicianship my students would possess. In any given class, I would have students who had had many years of formal training sitting alongside students who had no formal music training at all. How could I address the needs of the advanced students without leaving the musically untrained students in the dark?

Another difficulty I encountered was the mainstreaming of students with disabilities, mental or physical, in my classes. How were my special needs students going to play a scale on the keyboard, let alone understand one, if they had difficulty hitting the glockenspiel keys with a mallet? These concerns, I soon discovered, were common concerns among educators. I began to look toward technology for assistance.

It was during this time that I first thought of creating some adaptive instruments that would help bridge the gaps among the musicianship levels of my students. There were musical concepts like harmony, scales, and dynamics that I wanted to teach that I might never cover to the degree that I wanted if I continued to spend most of my class time explaining proper glockenspiel grip and technique.

EAMIR

The first adaptive instrument I used with my students was created in Max and involved some objects to track the location of a specific color from a video camera and generate numbers for the position of the color on the screen. The position of the color from left to right generated the numbers 0–127 from low to high, while the position of the color from top to bottom generated the numbers 0–127 from high to low. To produce a consistent color, I used a laser pointing device.

We will go into more depth about this patch in Chapter 16 when we discuss working with live video. In essence, the patch was largely based on taking the horizontal numbers (0–127) and filtering them to a diatonic scale using the *coll* process outlined earlier while using the vertical numbers to determine velocity values.

In my music classroom, I set up the software on a computer and connected a webcam. I had a student hold a laser pointing device in front of the webcam. As she waved the laser from side to side, the software tracked the color of the laser and played through the pitches of the C Major scale from low to high at different velocities. The student needed only to hold the pointing device for the software to track the color from the laser pointer and create diatonic pitches.

Soon I had some of the more musically proficient students providing an accompaniment on traditional instruments like drums, bass, and guitar in

the same key as the "laser" patch. At that point, I was able to discuss concepts of high to low pitch as well as varying dynamics. To make the situation better, I had my students demonstrate knowledge of these concepts by performing them: the more advanced students with their traditional instruments, and the untrained and special needs students with this accessible software instrument.

A few weeks later, I designed a few more patches based on the same pitch filtering concepts only using different control devices. I started working with what I had around me by using the computer's mouse and keyboard to trigger notes. Next, I started using a touch-screen computer to trigger notes; then, graphics tablets (borrowed from our art department) and Smartboards. For each new patch, I was able to reuse almost all of the Max code except the parts that received numbers from the different control devices; each patch basically accomplished the same thing using a different control device.

The idea of making tonal music from lots of different devices was intriguing to me. I began buying sensors (such as those from *Electrotap)*[3] and putting them all around my classroom. I put pressure sensors on the walls, doors, and floors; light sensors by the windows; anything that would give me some numbers that I could map to music. I started referring to my classroom as the *Electro-acoustic Musically Interactive Room,* or *EAMIR,* for short.

Soon afterward, I began putting these applications on the Internet at www.eamir.org so my students could freely download them. Not only did the students download them, but many of their parents did as well. Using these instruments was a great way to supplement teaching the musical objectives I was trying to convey to my students, and the novelty of using technology in this way got their parents involved in the process.

Control Interfaces

A variable is something that changes. A control is something that changes a variable. In music, there are many variables, such as pitch, dynamics, and timbre that change as a result of the instrument's control device, also known as a control interface.

The control interface for a violin is typically a bow. Without buttons, knobs, or sensors, the bow is capable of controlling numerous variables within a single, simple, interface. For example, if you angle the bow differ-

3. Electrotap sensors and interfacing products designed to work with Max are available from http://www.electrotap.com/. Visit the Cycling '74 webpage, www.cycling74.com for other types of sensors and hardware that can be used with Max.

ently as it hits the strings, the timbre will change; apply more pressure and the dynamics will change.

The Buchla 200e (see Figure A) is a modular synthesizer also capable of controlling numerous musical variables. In fact, the Buchla is capable of creating more diverse timbres than the violin. However, controlling musical variables on the Buchla, with a control interface of knobs, buttons, and patch cables, involves more gestures than the violinist and the bow.

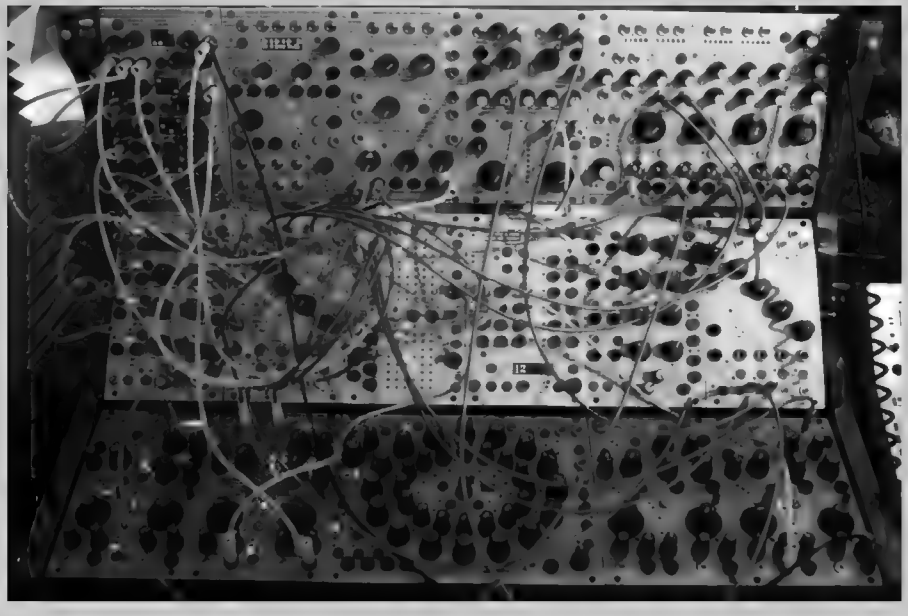


FIGURE A

the Buchla 200e created by Donald Buchla (Tiemann, 2006)

For the intent of performance, some control interfaces are more accessible than others for real-time use. With a computer, you can arguably achieve any sound imaginable if you tweak the right numbers and press the right buttons. It is a well-designed control interface, however, that allows a performer to readily control musical variables in a less cumbersome way than clicking on menu items from pull-down lists and checking boxes.

Throughout history, people have created new musical instruments, and the instruments created generally reflect the technological resources available at the time. Early primitive instruments had few moving parts, if any. The Industrial Revolution made way for the modern piano to evolve using steel and iron. In the Information Age, it stands to reason that newly created instruments may largely involve computers and electronics.

New Interfaces for Musical Expression (NIME)[4] is an international conference in which researchers and musicians share their knowledge of new instruments and interface design. Session topics include controllers for

4. New Instruments for Musical Expression http://www.nime.org/

performers of any skill level as well as the pedagogical implications of using these controllers.

Tod Machover, known for many great technological contributions to music including the *Hyperinstruments*[5] group, shared an interesting thought: "Traditional instruments are hard to play. It takes a long time to [acquire] physical skills which aren't necessarily the essential qualities of making music. It takes years just to get good tone quality on a violin or to play in tune. If we could find a way to allow people to spend the same amount of concentration and effort on listening and thinking and evaluating the difference between things and thinking about how to communicate musical ideas to somebody else, how to make music with somebody else, it would be a great advantage. Not only would the general level of musical creativity go up, but you'd have a much more aware, educated, sensitive, listening, and participatory public" (1999).

With proper practice, an individual can control most variables of an instrument well and at very fast speeds. However, the initial performance accessibility of an instrument or control interface has definite implications for its use by individuals as a musical instrument—in particular, those individuals who lack formal musical training and those who have physical or mental impairments.

Videogame controllers are typically designed with mass accessibility in mind. Many early videogames used joysticks and only a few buttons to control game play. Game controllers today are typically comfortable to hold with buttons and other switches positioned to allow the user to access them easily. Some developers of controllers such as Nintendo have incorporated sophisticated sensors and gyroscope technology into their controllers to provide continuous data about the controller in addition to using buttons and switches. A few EEG-based game controllers exist that aren't held but worn on the head and controlled by measuring brain waves.

After I had created a few EAMIR patches, I began using the same basic program functionality with different types of videogame controllers. In one lesson, one of my teaching objectives was to discuss things about functional harmony and how certain diatonic chord progressions functioned.

In a classroom of individuals with mixed levels of musicianship, I didn't want to spend the majority of my time focusing on how to play each chord in the progression on their instrument—that was something they could work on with their instrumental teacher, and some students didn't have an instrument. I wanted to discuss the chord itself and how it functioned among other chords in a given key. I wanted them to hear and experience

5. Hyperinstruments/Opera of the Future http://www.media.mit.edu/hyperins

the concept I was trying to explain and I didn't want them to miss it because of their lack of ability to demonstrate the concept themselves on a traditional instrument.

At that time, the game Guitar Hero was becoming extremely popular. All of my students played the game and were familiar with its unique guitar-shaped controller. Naturally, I felt that using this controller to trigger chord functions would provide some of my students with an accessible control interface to perform with that they already knew how to use while allowing me to discuss my teaching objectives: chord functions. Plus, it's really cool looking!

In Max, I made a patch that mapped each of the first 4 buttons, in combination with the two-position flipper, to the 8 chord functions of a given key. Button one pressed down with the flipper held down triggered the one chord, button one held down with the flipper held up triggered the two chord, and so on. The fifth button, in combination with the flipper, triggered one of two chord voicings: one that played a full root position "rhythm" triad and another that played only the single root note of each chord in a higher "lead" octave. The controller then had the ability to play chord functions and scales in any diatonic key starting on any tonic.

Since the controller was already familiar to the students, there was little instruction needed to explain how the instrument worked, and we were soon able to discuss chord progressions in terms of chord functions. Imagine my surprise when one of my students who couldn't (or wouldn't?) play a glockenspiel was able to tell me that he liked the sound of a "I V vi IV progression in E Major." Others enjoyed hearing how chord functions differed in more distant tonalities like *Harmonic Minor* and *Lydian b7*. I even made some specialized notation for the activity using the "colored noteheads" feature in the notation software we used in which the arrow markings indicate which position to "strum" with the flipper.

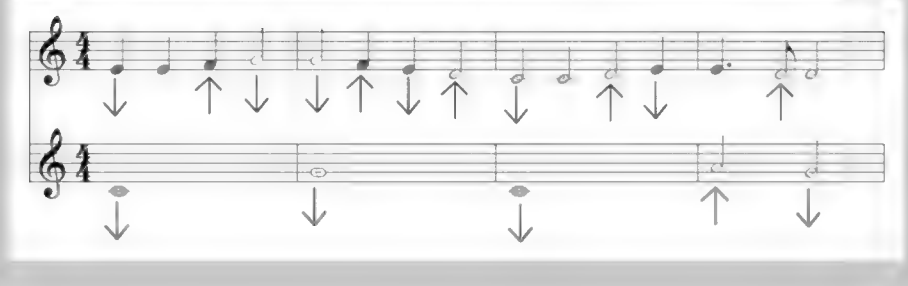


FIGURE B

Guitar EAMIR-O notation with colored noteheads and flipper position markings

Students soon began bringing in other game controllers and interfaces to see if they could make music with those as well. All the while, the patch I modified to suit these different controllers remained the same in most

ways. The only part that needed to be rewritten was the part of the patch that mapped data from the buttons of whatever game controller I was using to the buttons in my patch that triggered chord playback.

Anything that Max can get data from is fair game for use as a control device. There are even applications for mobile devices that can send data to Max with relative ease. You can also use mobile apps to control your computer's mouse and keyboard, thereby sending data to Max. Wireless mice, keyboards, and numeric keypads can be used as inexpensive controllers or modified to become footswitches. In the next chapter, we will discuss how to get data from videogame controllers. Remember: it's cool to say that you got your phone or your joystick to communicate with a Max patch, but it's all about how you map that data that really matters.

Chords

Before we look at the next EAMIR SDK example, we should discuss another third party object that serves as a shortcut for generating chords: the *modal_triad* object. As you know, when you installed the EAMIR SDK, you also installed the Modal Object Library. In fact, there's also a menu item for the Modal Object Library in the Extras menu.[6]

1. Click on *Extras>Modal_Object_Library* to view the main menu of the Modal Object Library

Among the objects and abstractions available in this library are the *modal_change* object we discussed last chapter as well as the *modal_triad* object I just mentioned.

2. Click on the *message* box containing the text *modal_triad* to open the *Help* file for this object

The *modal_triad* object functions similarly to the "chord maker" patch we created in Chapter 4. It receives a set of scale degrees (presumably from *modal_change*, but not required), and when the object receives the numbers 1–8 in its first inlet, it outputs the notes for the corresponding chord function of the same scale. It also has a bunch of built-in shortcuts that allow you to send specific chord names in *message* boxes such as *C* for a C major chord, *V/5* for a key specific tonicization (V of V), or even *Gdom7b9#11*. A complete list of supported chord *messages* is listed in the *modal_triad* Help file.

6. If you did a manual install of the EAMIR SDK, you will need to locate the folder *EAMIR_SDK>EAMIR_3rd_Party_externals*on your computer and open the file *the Modal Object Library.maxpat.*

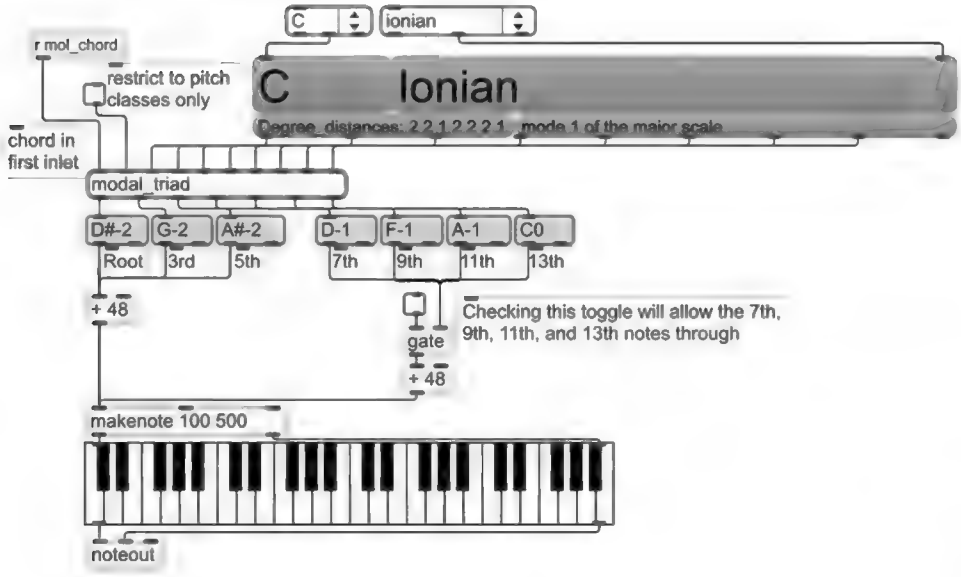


FIGURE 7.8

modal_triad Help file | modal_triad.maxhelp

The function of the *modal_triad* object is pretty straightforward and, as you probably realized, you already know the concept of creating chords this way. Like the *modal_change* object, using *modal_triad* will simply help us to speed up our implementation of chords throughout this book. Of course, you are under no obligation to write your patches using this, or any, third party object. You may alternatively refer to the technique we used to implement chords as described in Chapter 4.

Despite the "triad" part of the name, the *modal_triad* object has 7 outlets to output the 7 possible pitch classes of a given chord. As you can see in the Help patch, *modal_triad*receives chord *messages* in its first inlet wirelessly from the *r mol_chord* object. The first 3 outlets of *modal_triad*are connected to *number* boxes where *48* (4 octaves) are added to them before they are synthesized (*makenote, noteout*). The remaining 4 outlets are also connected to *number* boxes that are received by the second inlet of an object called *gate*.

The *gate* object allows you to restrict the flow of data received in its second inlet. The first argument of *gate* specifies the number of outlets you would like the object to have. If you want the option to send the incoming data to 3 different locations, then the first argument of *gate* would be *3*. The actual data is received in *gate*'s second inlet whereas a number sent to *gate*'s first inlet signifies which of its outlets data will be sent through. The number *1* received in *gate*'s first inlet would cause incoming data to leave through *gate*'s first outlet; the number *0* would cause the *gate* to be "closed" and, therefore, data output would be restricted from exiting any outlet. For this reason, a *toggle*, which, as you'll recall, sends out the numbers *1* and *0*, is commonly used as a control for *gate* objects with only one outlet. A *radiogroup, umenu*, and even *message* boxes are good choices to control *gate* objects with more than one outlet.

In this Help patch, the *gate* is used to restrict the number of outputted chord tones to 3. To let all 7 chord tones through, simply check the *toggle* which sends a 1 to *gate* allowing the remaining 4 notes to pass through *gate*'s outlet to the + *48* and the rest of the synthesis objects.

Close the Help file for *modal_triad* as well as the Modal Object Library *Extras* menu.

3. If it's not still open, click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
4. In the *umenu* labeled *Examples*, click the second item *2.EAMIR _Chord_Basics.maxpat*

As you can see, this example patch has 3 *bpatcher*s in it. The one on the bottom is the same "MIDI Out" *bpatcher* used in the previous example. The top *bpatcher* contains the *modal_change* and *modal_triad* objects to generate chords, while the *bpatcher* in the middle is used to control the voicing of whatever chord tones it receives. Unlock the patch and

5. Ctrl+click (Mac) or right click (Windows) the upper *bpatcher* and select *Object>Open Original "EAMIR_chord_generation. maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching mode.

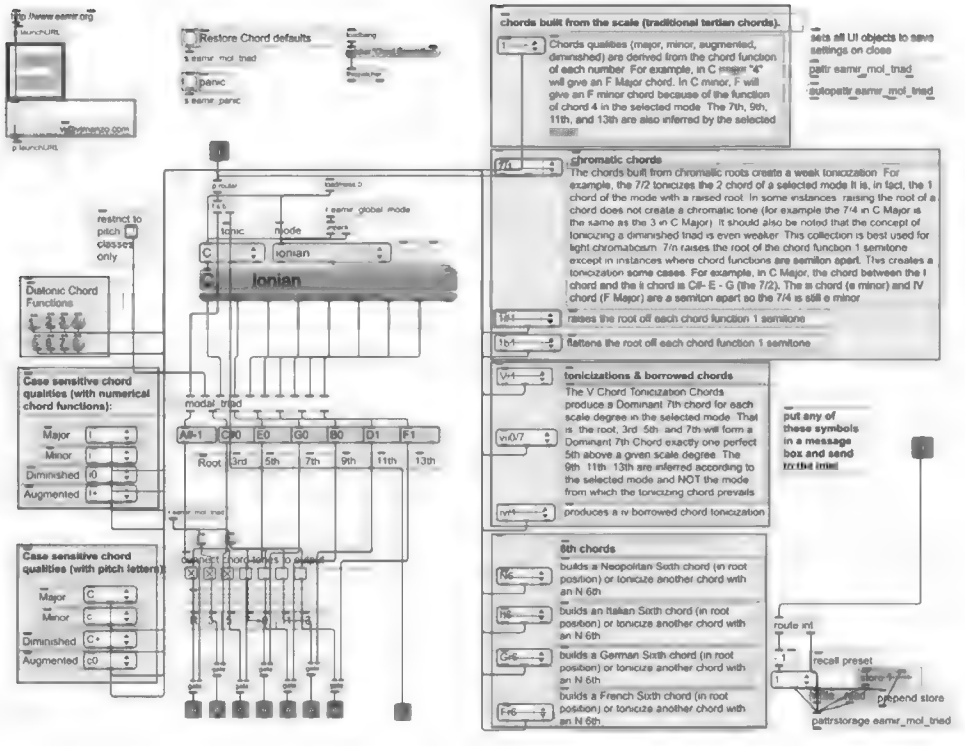

Restore Chord defaults
panic
chords built from the scale (traditional tertian chords).
Chords qualities (major, minor, augmented, diminished) are derived from the chord function of each number. For example, in C major "4" will give an F Major chord. In C minor, F will give an F minor chord because of the function of chord 4 in the selected mode. The 7th, 9th, 11th, and 13th are also inferred by the selected
sets all UI objects to save settings on close
pattr eamir_mol_triad
autopattr eamir_mol_triad
chromatic chords
The chords built from chromatic roots create a weak tonicization. For example, the 7/2 tonicizes the 2 chord of a selected mode. It is, in fact, the 1 chord of the mode with a raised root. In some instances, raising the root of a chord does not create a chromatic tone (for example the 7/4 in C Major is the same as the 3 in C Major). It should also be noted that the concept of tonicizing a diminished triad is even weaker. This collection is best used for light chromaticism. 7/n raises the root of the chord function 1 semitone except in instances where chord functions are semitone apart. This creates a tonicization some cases. For example, in C Major, the chord between the I chord and the ii chord is C#- E - G (the 7/2). The ii chord (a minor) and IV chord (F Major) are a semitone apart so the 7/4 is still a minor
raises the root off each chord function 1 semitone
flattens the root off each chord function 1 semitone
tonicizations & borrowed chords
The V Chord Tonicization Chords produce a Dominant 7th chord for each scale degree in the selected mode. That is, the root, 3rd, 5th, and 7th will form a Dominant 7th Chord exactly one perfect 5th above a given scale degree. The 9th, 11th, 13th are inferred according to the selected mode and NOT the mode from which the tonicizing chord prevails
produces a iv borrowed chord tonicization
put any of these symbols in a message box and send to the inlet
8th chords
builds a Neopolitan Sixth chord (in root position) or tonicize another chord with an N 6th
builds an Italian Sixth chord (in root position) or tonicize another chord with an N 6th
builds a German Sixth chord (in root position) or tonicize another chord with an N 6th
builds a French Sixth chord (in root position) or tonicize another chord with an N 6th
restrict to pitch classes only
Diatonic Chord Functions
tonic
mode
C
Ionian
Case sensitive chord qualities (with numerical chord functions):
Major
Minor
Diminished
Augmented
modal_triad
All-1
C#0
E0
G0
B0
D1
F1
Root
3rd
5th
7th
9th
11th
13th
Case sensitive chord qualities (with pitch letters):
route int
recall preset
store 1
prepend store
pattrstorage eamir_mol_triad

FIGURE 7.9

contents of bpatcher similar to modal_triad Help file | EAMIR_chord _generation.maxpat

Before you start thinking this patch is complicated, take a closer look. It's actually very similar to the Help file patch for *modal_triad* that we just discussed.

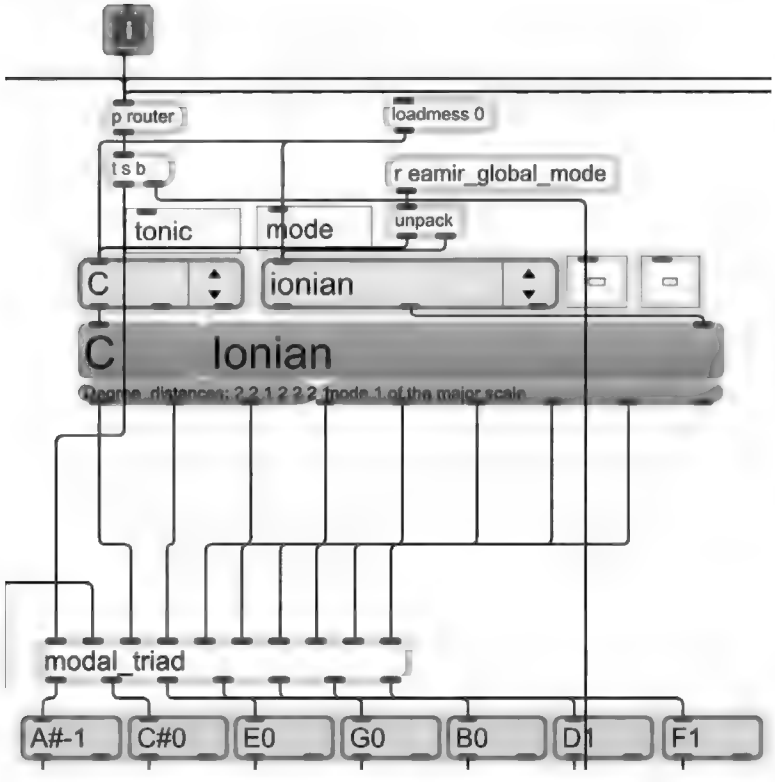


FIGURE 7.10
modal_triad and modal_change used to generate chords | EAMIR_chord_generation.maxpat

Chord names and numbers are received by this patch through an *inlet* object. The data arethen sent to a subpatcher called *p router*. Let's skip past this subpatcher for a moment and follow the data-flow to the next object, *t* (*trigger*).

Stay with me for this next part. Data are received in *t*'s inlet first causing a bang[7] to be sent from *t*'s last outlet, labeled *b*, and, second causing any *symbols* received by *t* to be sent from *t*'s first outlet. Symbols are just another data type, just like bangs, integers, and floating point numbers, that Max deals with. Where a list in a *message* box may contain a number of items in it, a symbol is one entity.

The *modal_change* can receive symbols and integers as messages to cause chord output. I could add another item in *t* for integers (*i*), in order to have one outlet for symbols like E♭ or D, and one outlet for numbers (1–8); however, when a *symbol* is received by *t,* a *0* is sent through the integer outlet of *t.* Obvi-

7. This bang will be sent to the patch's last *outlet* object labeled "sustain." Since it is sent right before the new chord is played, the bang will be useful in a main patch for triggering objects that release sustain such as the *flush* object as discussed in Chapter 4.

ously, there is no such thing as the *0* chord in a diatonic key, so *modal_change* won't know what to do with that message. To simplify the data going to *modal _change*, I've opted to use the subpatcher *p router* to convert any *numbers* received into symbols.

6. Lock the patch and double click *p router* to open the subpatcher

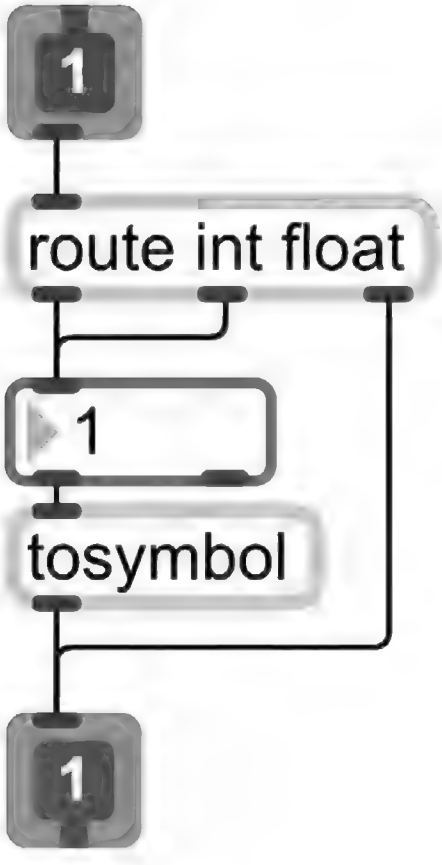


FIGURE 7.11

converts integers into symbols | subpatcher in EAMIR_chord_generation .maxpat

In this subpatcher, all data can be received by a single inlet regardless of whether the input is a *number* or a *message*. Any *int*s or *float*s will be sent to the *tosymbol* object (floating-point numbers, if any are received, will be truncated at the decimal point by way of the *number* box). Any data not a number will not be identified by *route* and, thus, will be sent from *route*'s last outlet. The *t* object will receive the newly converted numbers as symbols as well as the single *messages*.

In essence, these last few paragraphs depict one workaround for converting all data to a common type. The necessity for this was to let *modal_triad* receive the same data in one format from one source. The whole process is not a lot to write home about, but useful knowledge if you're going to be using Max, as situations like this tend to arise and need to be conquered methodically. Instead of panicking and quitting Max, it's best to think logically about the flow of data from one object to another, test dataflow to ensure that everything is getting where it needs to be, and read the Help files and Reference pages for information. Let's keep going.

The rest of the patch primarily consists of a bunch of different *umenu*s showing examples of chord names that can be used. Of course, users will be sending chord names to the inlet of this patcher once it is loaded within a *bpatcher* in a parent patch.

At the bottom of this patch is a series of *gate* objects that allows the user to control the number of chord tones that will be used. The *r eamir_mol_triad* object is used to reload the default chord tone configuration where only the root, third, and fifth are enabled.

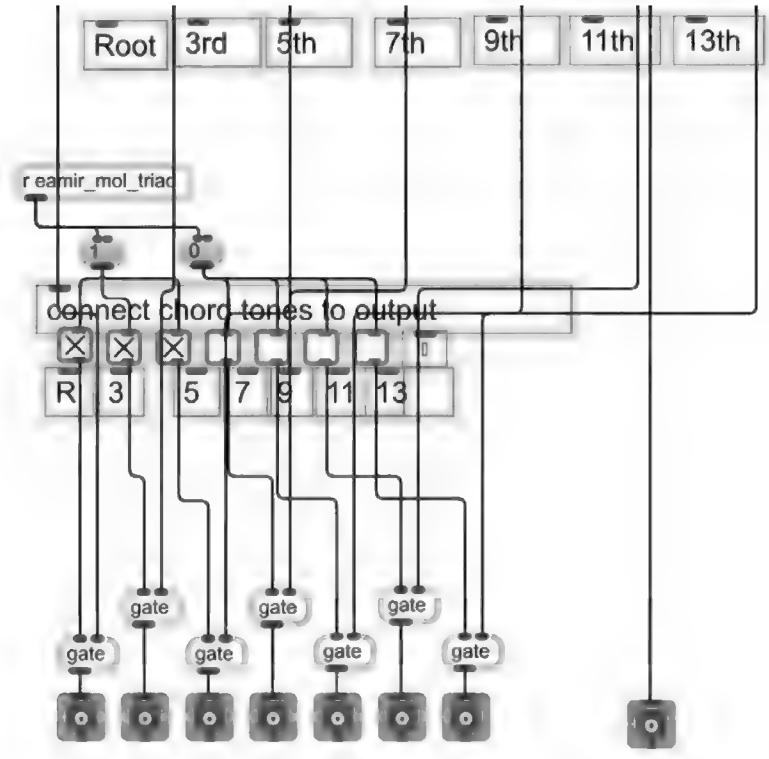


FIGURE 7.12
restricting the number of chord tones present by using gate objects | EAMIR_chord_generation .maxpat

Storing Presets

The last bit of this patch worth discussing is the "recall preset" portion of the patch in the lower right. Data to this patch are received through a second *inlet* object and allow a user to enter a number to recall a stored preset using the object *pattrstorage*.

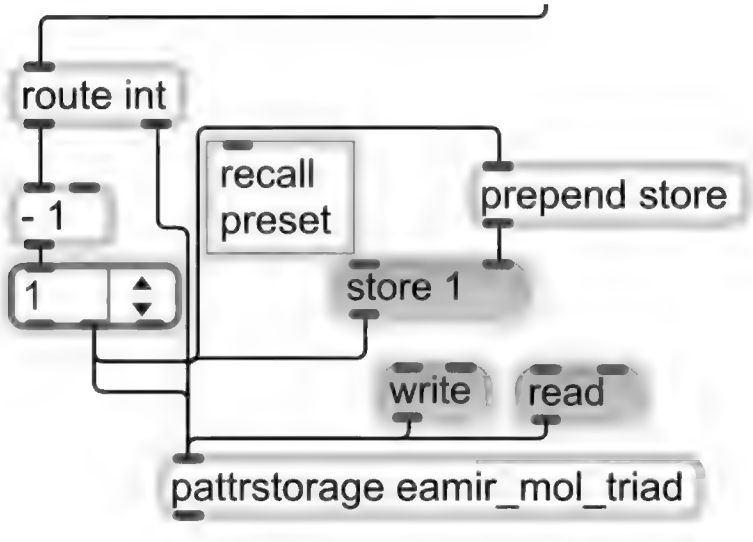


FIGURE 7.13
storing and recalling presets | EAMIR_chord _generation.maxpat

The *pattrstorage* object, like *pattr* and *autopattr*, takes a name as its argument to work in conjunction with *pattr* and *autopattr,* which are also located in

this patch. The *message store* and a number (ex. *store 1*) stores the state of each object included in *autopattr* (by way of the *Scripting Name*) into *pattrstorage* where it can be recalled later.

The *umenu* contains the numbers *1–100*. When a number is selected from the *umenu*, the number is sent from *umenu*'s inlet to the inlet of *pattrstorage* where the object looks to see if a preset has been previously stored at that number. Selecting the *umenu* also sends the number to the *prepend* object with the argument *store*.

The *prepend* object adds its argument to the beginning of anything that comes through its inlet. Because the *pattrstorage* object can receive the *message store 1*, the *umenu* item number is sent to *prepend store* in order to format the message into a *message* box. When a user clicks on the *message* box, the preset data are then stored at that preset number. Choosing the number from the *umenu* allows the user to recall that preset.

The *write* and *read* messages allow the user to save their presets into a file which can then be loaded on a different computer if desired. Close this patch without saving the changes.

In the *EAMIR_Chord_Basics.maxpat*, we can see that sending *message* boxes to the *bpatcher* generates the desired chords. The "Chord Voicing" *bpatcher* simply takes the pitch class for each chord tone and transposes it to a desired octave allow multiple iterations of the same note to be voiced in different octaves. This patch also contains preset storing capabilities.

New Objects Learned:

- *bpatcher*
- *midiin*
- *midiparse*
- *outlet*
- *hint*
- ror *receive*
- sor *send*
- *midiout*
- *midiformat*
- *pattr*
- *autopattr*
- *gate*
- *tosymbol*
- *pattrstorage*
- *prepend*

Remember:

- To save patch settings using *pattr*, create an object called *pattr* and give it a name as an argument. Create an object called *autopattr* and give it the same name as an argument. Open the *Inspector* for any object whose settings you want to include in the save, and enter the same argument name as its *Scripting Name*. Piece of cake!

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Remote Messaging—Sending messages without patchcords
  - Parsing—Decoding and encoding MIDI streams
  - Pattr Basics—Introduction to state management
  - Autopattr Bindings—Advanced patcher storage
  - Bpatchers—Working with inlined patchers

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows you to play the diatonic chord functions for a user-selected key by using the number keys 1–8 on your computer keyboard. HINT: consider using *modal_triad* and *modal_change* or the EAMIR SDK *bpatcher*s to speed up your process. You will also need some way to select certain numbers from your computer keyboard. Use UI objects like *slider* to control variables and make sure that these settings automatically save in the patch by using *pattr* and *autopattr*. Use *pattrstorage* to save a few presets for your patch.



Control Interfaces Continued

In this chapter, we will look at some innovative ways to control music making as we develop musical instruments. We will look at using your computer keyboard and mouse as performance instruments as well as discuss the use of videogame controllers in your patches. Designing your own custom musical instruments is a great way to tailor the controls to the specific physical abilities of users while allowing them to focus on certain specific musical concepts like pitches, scales, and harmony/chords.

1. Click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
2. In the *umenu* labeled *Examples*, click the third item *3.EAMIR _ASCII_Keyboard_Control.maxpat*

Unlock the patch that opens and look at its basic structure. As you can see, the patch is really just 4 *bpatcher* objects, 3 of which refer to patches we've already looked at. The newest *bpatcher*, at the top of the patch, is basically just a patch with a *key* object, a *select* object, and some fancy graphics—all things you learned to use in Chapter 3. Lock the patch and

3. Type your full name using your computer keyboard. Note that uppercase letters and lowercase letters trigger different *buttons*
4. Press the number keys 1–8 as these are mapped to *message* boxes containing numbers used as diatonic chord functions

Without the top *bpatcher*, your patch generates chords in any key simply by clicking the *message* boxes. The top *bpatcher* is just a control interface that maps something (keys) to something else (*message* boxes).

5. Ctrl+click (Mac) or right click (Windows) the top *bpatcher* and select *Object>Open Original "EAMIR_keyboard.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching view.

The contents of the patch are as I described: a *key* object, as well as a *keyup* object, are connected to two gigantic *sel* (*select*) objects containing the ASCII numbers for all the available characters on the computer keyboard—nothing you couldn't already do. In fact, the most impressive part of this patch, in my opinion, is the graphical part of it. Remember the wise words of David Cope: "many times, programs are 1% function and 99% window dressing."

Arguments for Abstractions

You may be wondering about this object called *eamir_kb_button*. Well, it's actually an abstraction like we mentioned earlier. The object box is referring to a Max patch called *eamir_kb_button.maxpat* located inside the EAMIR SDK folder you installed. Each instance of this abstraction in the patch is separate from the next because they all have different variables. For example, lock the patch and

6. Double click on the first instance of this abstraction beneath the *sel* object called *eamir_kb_button ~ `*

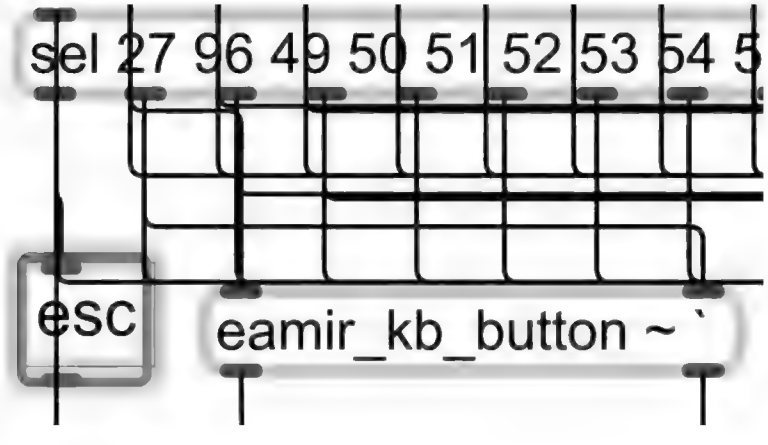


FIGURE 8.1
sel object connected to the abstraction eamir_kb_button | EAMIR_keyboard.maxpat

When the window opens, notice that the two *message* boxes contain the characters ~ and `, respectively.

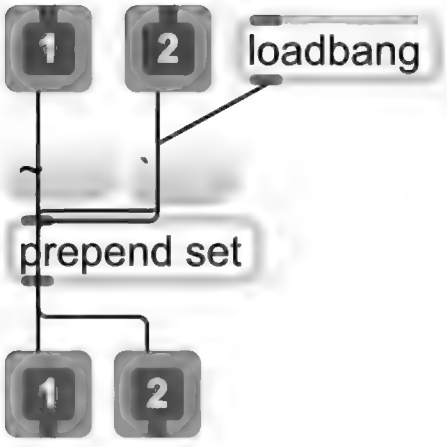


FIGURE 8.2

message boxes connected to prepend object | abstraction eamir_kb _button.maxpat

7. Close that window and double click the instance of this abstraction directly to the right of *eamir_kb_button ~* ' called *eamir_kb _button ! 1*

Notice that in this window, the *message* boxes contain *!* and *1,* respectively. This doesn't mean that I have a whole bunch of different versions of the *eamir_kb _button* patch, each with a different combination of two arguments. Though I could have done that, the process would take forever to complete and would be comparable to some form of mental torture. Instead, let's look at the shortcut that I took to make this process more efficient. Unlock the patch and

8. Ctrl+click (Mac) or right click (Windows) on any *eamir_kb _button* object and select *Object>Open Original " eamir_kb _button "* from the contextual menu

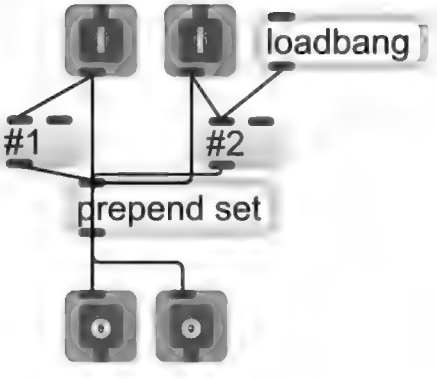


FIGURE 8.3

message boxes contain #1 and #2 as placeholders for arguments | abstraction eamir_kb_button .maxpat

The patch that opens shows the secret to this efficient method: instead of "hard coding" each keyboard character into the *message* boxes of the *eamir_kb_button* patcher, the *message* boxes contain the characters *#1* and *#2*. These are placeholders for arguments given to object name. When the *eamir_kb_button* object is given, for example, the first argument *Q* and the second argument *q*, the *Q* replaces the first placeholder, #1, and the *q* replaces the second placeholder, #2. In this way, we can write the abstraction in an open-ended fashion so that it can be reused in a number of different ways simply by changing the arguments. If we had specified #3 and #4, the *eamir_kb_button* could take up to 4 arguments. This process is a huge time saver.

The rest of this abstraction is pretty straightforward. The output of this abstraction is a *comment* box that will display either the uppercase or lowercase character depending on what was pressed by the user. The *loadbang* sends the second of the two arguments by default. The *prepend set* takes the *message* and sets it into the *comment* box replacing whatever text was already present.

Another part of interest in this patch is the option to switch the way that a bang is sent from the keys—either when the key is pressed or when it is released. A simple *radiogroup* object allows the user to choose the "keyboard mode": "key down" or "key up."[1] Data are sent wirelessly through the *s eamir_keyboard_button_mode* object to subpatchers named *p pad_mode* at the far right of the patch.

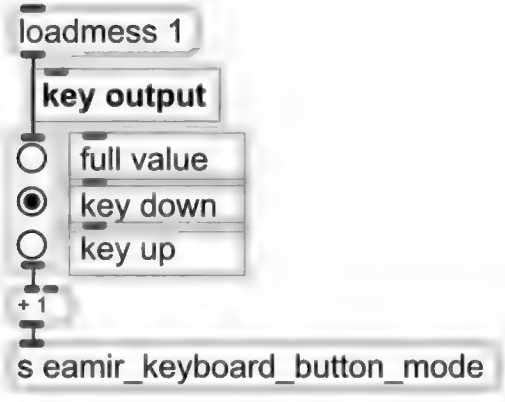


FIGURE 8.4

radiogroup sends data wirelessly through send object | EAMIR_keyboard .maxpat

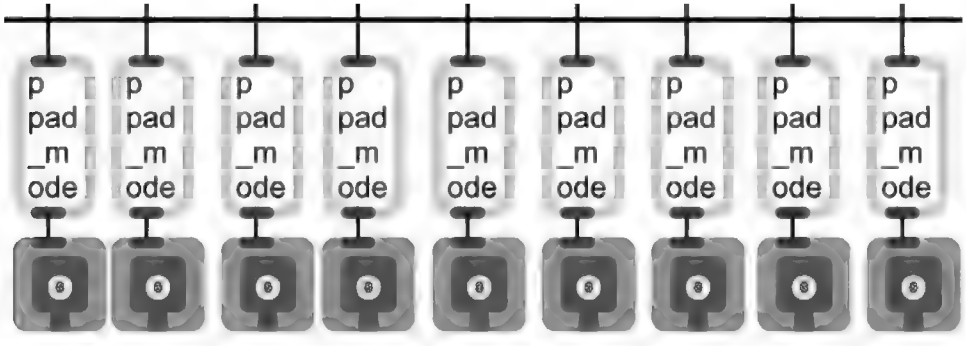


FIGURE 8.5

subpatchers connected to outlets | EAMIR _keyboard.maxpat

The selected "keyboard mode" will result in data from one of 3 sources being sent to the *outlet* by way of the *switch* object.

1. The "full value" option is a third mode of *radiogroup* allowing velocity sensitive keys to be used. Since the ASCII keyboard is not touch sensitive, this mode does not apply for this patch, though the option would be useful if a touch sensitive controller were used.

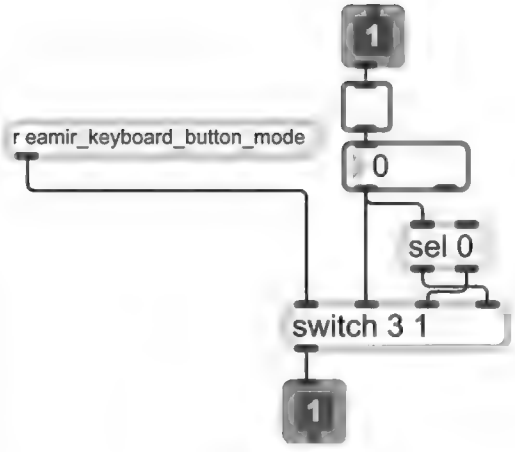


FIGURE 8.6

switch receives data from radiogroup wirelessly to specify which input will be sent to the outlet | subpatcher in EAMIR _keyboard.maxpat

The *switch* object takes data in any number of inlets; the number of inlets is defined by *switch*'s first argument. The first inlet of *switch*, however, is reserved for numbers that determine which of the inlets will be allowed to pass through to the outlet. In some ways, *switch* is like *gate*, but switch has multiple inputs and one output, whereas *gate* has one input and can have *multiple* outlets.

When a key is pressed on the computer keyboard, the ASCII number is sent from *key* and compared to the arguments of the *sel* object where a bang is sent from one outlet of the *sel* object and received by the *inlet* of a *p pad_mode* subpatcher. Inside the subpatcher, the bang turns the *toggle* "on." When the key is released from the computer keyboard, the *keyup* object follows the same actions as the *key* object described earlier and a second bang is received within the subpatcher turning the toggle "off."

The *sel 0* object within the subpatcher will detect a *1* when the *toggle* is turned on, that is, when the key is pressed, and a *0* when the *toggle* is turned off, that is, when the key is released. The number sent to *switch*'s first inlet from the *radiogroup* will determine which of the inputs received by *switch* will be sent from its outlet.

Using the Mouse

We've worked with your computer keyboard as a musical instrument, so now, let's get your mouse involved in the music making.

1. Create a new patch
2. Create a new object called *mousestate*
3. Create a *toggle* and 2 *number* boxes beneath *mousestate*
4. Connect the first outlet of *mousestate* to the inlet of *toggle*
5. Connect the second and third outlets of *mousestate* to the inlet of the 2 *number* boxes, respectively
6. Create a *button* above *mousestate*
7. Connect the outlet of *button* to the first inlet of *mousestate*

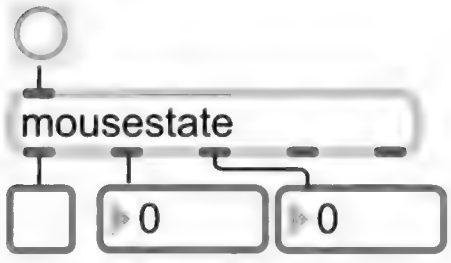


FIGURE 8.7
mousestate connected to number boxes | mouse_pitch.maxpat

Lock your patch. The *mousestate* object reports the location of your mouse when it receives a bang in its inlet.

8. Click on the *button* and note the values received in the *number* boxes

The values received in the *number* boxes relate to the horizontal and vertical position of your mouse with respect to your computer's monitor resolution. It will be helpful to see the numbers change as your mouse moves, but to do so we'd need to keep clicking the *button* object to report the position values. However, whenever we click the *button* our mouse is in the same position. Any thoughts?

Let's use a *metro* to continually click on the *button* for us. Unlock your patch and

9. Create a new object called *metro* with the argument *20* above the *button*
10. Create a *toggle*
11. Connect the outlet of *toggle* to the first inlet of *metro 20*
12. Connect the outlet of *metro 20* to the inlet of *button*

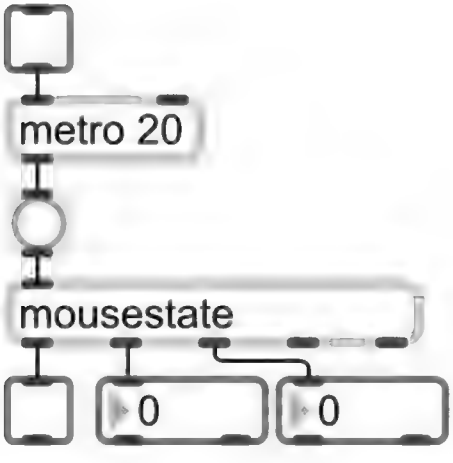


FIGURE 8.8
metro sends bang to mousestate every 20 ms reporting the mouse position | mouse_pitch .maxpat

Lock your patch click on the *toggle* to turn the *metro 20* on. As you move your mouse, you will see the different position values reported in the two num-

ber boxes, horizontal and vertical, respectively. If you click with your mouse, you will see the *toggle* blink.

13. Move your mouse to the absolute upper left corner of the screen and note that the number boxes read *0* and *0*. This is the position of your mouse with respect to your computer's screen resolution

The maximum horizontal and vertical values will vary depending on your computer hardware. To find out your maximum horizontal and vertical values,

14. Move your mouse to the absolute bottom right corner of the screen
15. Note the highest horizontal value and the highest vertical value as shown in the *number* boxes

It would great if we could somehow map these numbers to pitches and velocity so that depending on where your *mouse* was on the screen, it would play different notes. We could simply connect these numbers to a *makenote*, but the values would be way past the 127 limit of MIDI.

We can identify the range of values for our mouse movement. For example, if your maximum horizontal value was *1919*, your horizontal range is from *0* to *1919*. If your maximum vertical value was *1199*, your vertical range is from *0* to *1199*. We can use the *scale* object to map one range of numbers to another. We can create a *scale* object with the arguments *0 1919 0 127* to map the first range of numbers, the horizontal range, to the second range of numbers, the MIDI range.[2]

The *scale* object will do all the math for us to ensure that when our horizontal position is at *0*, the number output is *0*, and when our horizontal position is at *1919*, our number output is at 127. Equally important, it accurately maps all the numbers in between. Unlock your patch and

16. Create a new object called *scale* with the following for arguments: *0 1919 0 127* (Note: substitute *1919* for your maximum horizontal value)
17. Connect the first outlet of the *number* box receiving from *mousestate*'s second outlet, to the first inlet of the *scale* object
18. Create a *number* box
19. Connect the outlet of *scale* to the inlet of the *number* box

2. A typical pair of screen resolution values would be *1920* × *1200*. However, some Windows machines often report values that are "off by one," so the values *1919* × *1199* are used in this example. The values you use will depend on your screen resolution settings determined by your computer.

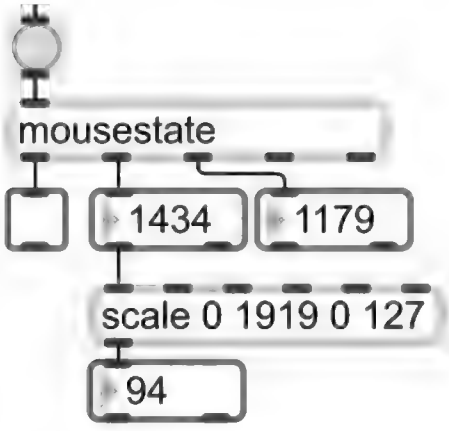


FIGURE 8.9
screen resolution horizontal value mapped to the MIDI range (0-127)| mouse_pitch.maxpat

Notice that as you move your mouse, the horizontal position numbers are "scaled" to the MIDI numbers. Let's synthesize this.

20. Create a new object called *makenote 100 300*
21. Connect the first outlet of the *number* box receiving from *scale* to the first inlet of *makenote 100 300*
22. Create a new object called *noteout*
23. Connect the outlets of *makenote 100 300* to the first 2 inlets of *noteout*

Now, as you move your mouse from left to right horizontally, MIDI notes will be played from lowest to highest. Let's map your vertical mouse position to control velocity so that when your mouse is at the top of the screen, the notes will be at their loudest.

24. Create a new object called *scale* with the following for arguments: *0 1199 127 0* (Note: substitute *1199* for your maximum vertical value)
25. Connect the first outlet of the *number* box receiving from *mousestate*'s third outlet, to the first inlet of the newly created *scale* object
26. Create a *number* box
27. Connect the outlet of *scale* to the inlet of the *number* box
28. Connect the first outlet of the *number* box to the second inlet of *makenote 100 300*

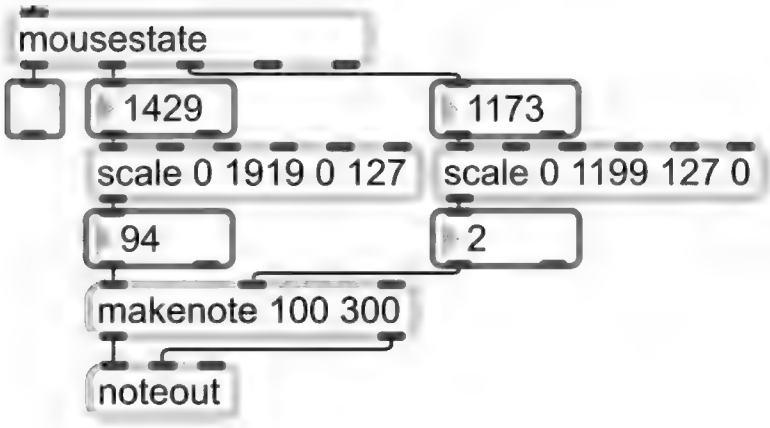


FIGURE 8.10

screen resolution horizontal value mapped to pitch; vertical mapped to velocity | mouse_pitch .maxpat

29. Move your mouse to generate notes

Now your mouse controls both pitch and velocity. Notice that we had to reverse the MIDI range for the second *scale* to *0–127* to allow a higher vertical position on the screen to yield a higher MIDI number. Save this patch as *mouse_pitch .maxpat* and close the patch.

Example 5 in the EAMIR SDK has a patch similar to the one we just created.

1. If it's not still open, click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
2. In the *umenu* labeled *Examples*, click the fifth item *5.EAMIR _mouse_control.maxpat*

This example is similar to the patch we just created though it is organized into separate *bpatcher* modules.

3. Move your mouse to generate notes
4. Turn off the mouse-to-MIDI mapping by unchecking the "on/off" *toggle* in the *bpatcher* labeled "horizontal value scaled to pitch"

The topmost *bpatcher* simply obtains the maximum horizontal and vertical values from the user's screen and outputs them to the other two *bpatcher*s that use *scale* to map the position values to some other set of numbers.

5. Ctrl+click (Mac) or right click (Windows) the top *bpatcher* and select *Object>Open Original "EAMIR_mousestate.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching mode. Notice how the maximum screen resolution values are reported using the *screensize* object. Close this patch.

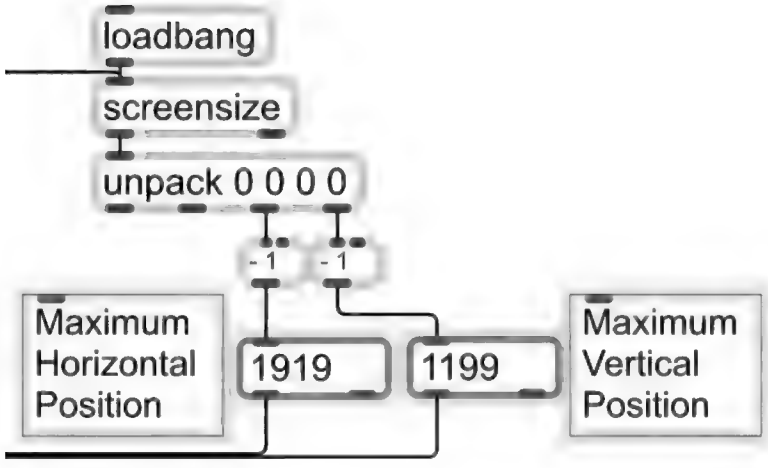


FIGURE 8.11

screen size values sent to scale objects | EAMIR _mousestate.maxpat

6. Ctrl+click (Mac) or right click (Windows) the middle-left *bpatcher* labeled "horizontal value scaled to pitch" and select *Object>Open Original "EAMIR_scaler.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching mode.

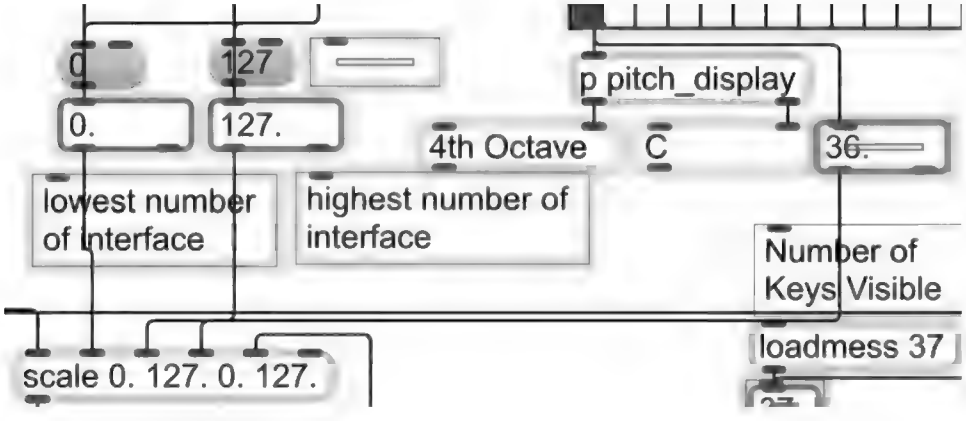


FIGURE 8.12

template for using the scale object | EAMIR _scaler.maxpat

The *scale* objects receive maximum screen resolution values from the patch's last two *inlet* objects. These *inlet* objects connect directly to the *scale* object changing its first two default arguments to the screen size. The two *kslider* objects are used to send two MIDI numbers, a low one and a high one, to the *scale* object replacing its last two default values. In this way, the "scaling" is customizable to any range of MIDI values. Close this patch.

The velocity *bpatcher* to the right works similarly to the previously discussed *bpatcher.* The "MIDI Filtering" *bpatcher* is basically the *coll* filter we used to make "random tonal music" in Chapter 6 with one added feature: remove repeated notes. Let's look at how this is accomplished.

7. Ctrl+click (Mac) or right click (Windows) the lower-left *bpatcher* labeled "pitch filtering" and select *Object>New Patcher with Contents of "EAMIR_filtering.maxpat"*[3] from the contextual menu

The contents of the patch *"EAMIR_filtering.maxpat"* will be copied into a new, blank patch that is unlocked and in Patching mode.

This patch is just like the one we made in Chapter 6 with the addition of the *change* object. The *change* object lets only a number different from the previous received number pass through. If you need to eliminate repetitions of the same number, the *change* object is a great choice. By using a *switch*, users can choose to allow all notes, including repeated notes, to pass through to the outlet, or use the *change* object to remove repeated notes. The latter option is sometimes useful for the type of pitch filtering we've been doing where a chromatic note becomes the nearest diatonic note thus producing a duplicate of the same diatonic note.

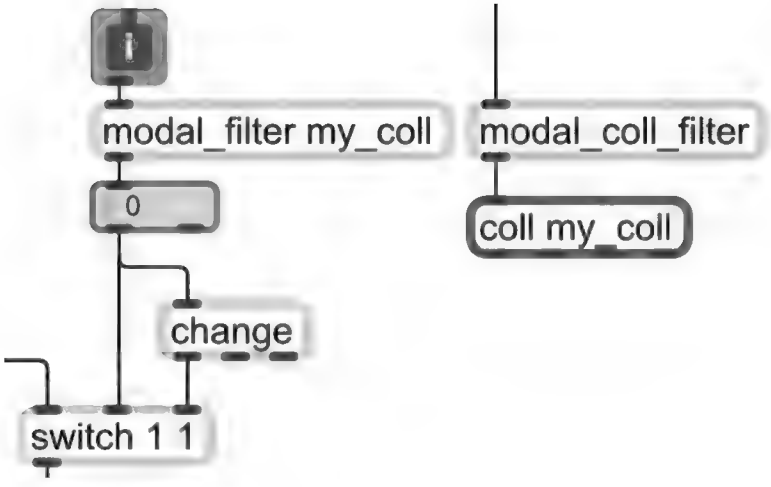


FIGURE 8.13
incoming pitches are filtering to a diatonic scale using a coll | EAMIR _filtering.maxpat

A few things about these EAMIR SDK patches are worth pointing out. First, these patches are mostly just modular versions of things you really already know how to do. They're just more organized with the use of *bpatcher* objects, which makes reusing parts of them a lot more accessible.

Second, particularly if you're creating an instrument, once you have the "what does it do" in order ("it plays chords," or "it plays two octave diatonic notes," etc.), you should focus on the "how does it do it" ("it uses keys on the ASCII keyboard," "it uses the mouse to play notes").

3. The reason the menu reads "*New Patcher with Contents . . .*" and not "*Open Original . . .*" is because the option "Embed in Parent Patcher" is checked for this *bpatcher*. This makes the patch the *bpatcher* references become part of the parent patch. Changes to the original embedded patch are then no longer reflected in the *bpatcher* patch.

Using Videogame Controllers

Let's discuss how to obtain data from videogame controllers and other human interface devices. Again, the control interface should be something that helps make using the patch more accessible. For this reason, I find that it's best to think about the audience of people that will be using your software and decide on a control interface that will work for them. If you feel that the mouse and keyboard are insufficient for the task, perhaps you can justify taking a trip to the local videogame store.

The *hi* object is useful for obtaining data from what your computer calls "human interface devices." These are often, but not limited to, USB game controllers such as joysticks and other controllers created for games to be played on a PC or Mac. If you have any of these devices connected to your computer and working (many times drivers must be installed), the data sent to the computer when the controller's buttons and toggles are pressed can be seen and used in Max. Keep in mind that these data will just be numbers. The videogame software designed to use these controllers will interpret these numbers in some way and map them to manipulate the game play. In the same regard, we can take these numbers in Max and manipulate them to control our patches.

Since it is unlikely that you and I have a common game controller available to us, I will explain the concept of obtaining data from a game controller using the *hi* object.

1. Create a new patch
2. Create a new object called *hi*
3. Open the Help file for *hi*

In the Help file, you can see a number of *message* boxes connected to the inlet of the *hi* object.

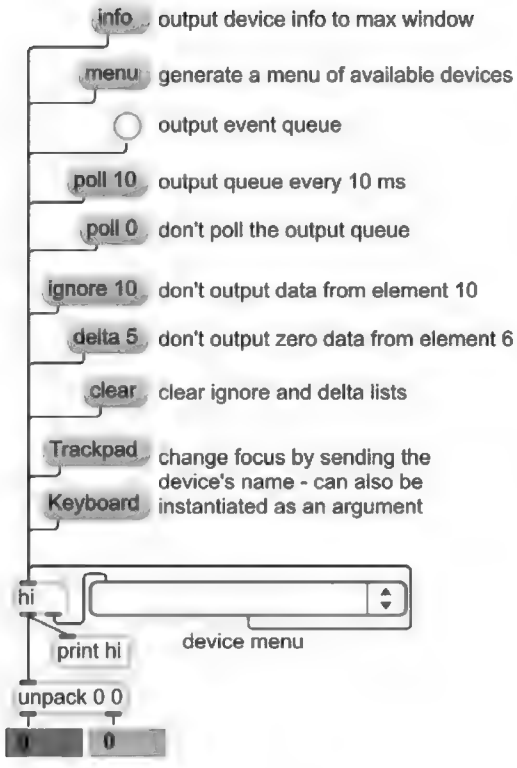


FIGURE 8.14

Help file for hi object | hi.maxhelp

The first outlet of *hi* outputs the data from a selected device. In the Help file, the data will be sent to an *unpack* object as well as to a *print* object. The second outlet of *hi* displays device names.

4. Click on the *message* box containing the text *menu* in order to populate a list into the *umenu* of all human interface devices connected to your computer

Depending on what type of computer you have, your keyboard and trackpad may show up in this *umenu* as well. For our demonstration, I have connected a USB joystick to my computer.

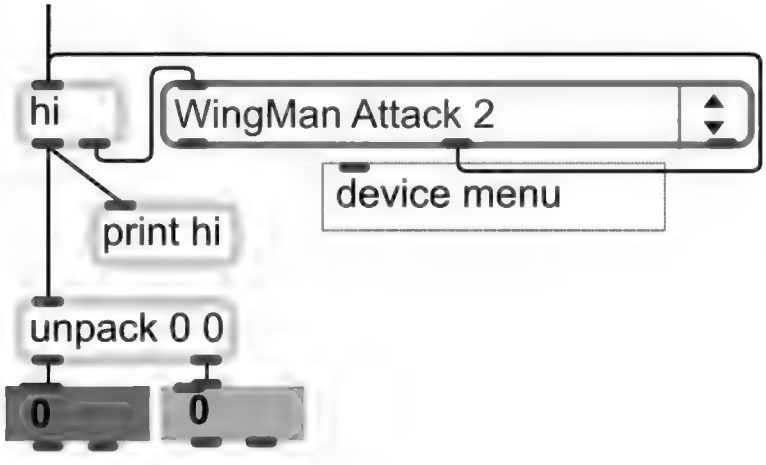


FIGURE 8.15

available human interface devices listed in umenu | hi.maxhelp

When I click the *menu* message, my joystick, called the *WingMan Attack 2*, populated into the *umenu*. If I select this item from the *umenu*, the *hi* object will focus on this device and the Max window will confirm by displaying the text from *print*: "hi: focusing on WingMan Attack 2."

To obtain data from the device, tell *hi* to "poll" for data every 10 milliseconds by clicking the *message* box containing the text *poll 10*. As you press buttons on the device and move any joysticks, you will see numbers fill the two *number* boxes and the Max window.

The next question is likely "How do I interpret these numbers?" It's fairly easy. Look at the Max window and press a button on the device; scroll down to the bottom of the Max window if necessary to read the numbers. A two-number pair will be displayed. Similarly to *coll,* the first number is like the index or address of the data while the second number, like the element, is the actual state or value of the data.

For example, if I press and release a button on the device, I may get the numbers *48* and *128* when I press the button and the numbers *48* and *0* when I release the button. In this case, the index would be *48* and the element would be *0* or *128*. Once you know the index of your data, you can then connect a *route* object with the argument *48* to the first inlet of *hi* to easily obtain just the element. This will provide an outlet from the *route* object that displays only data with the address *48*. Unlock your patch and

5. Create a new object called *route* with argument *48* (Note: substitute *48* for the data index you want to obtain data from)
6. Connect the first outlet of *hi* to the inlet of *route*
7. Create a *number* box
8. Connect the first outlet of *route* to the inlet of the *number* box

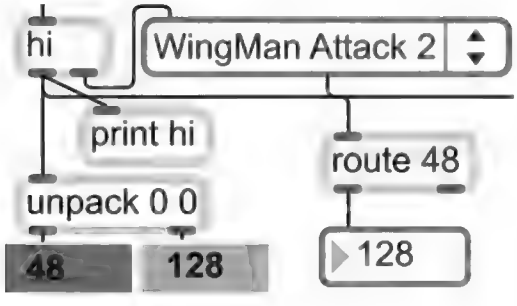


FIGURE 8.16
data from hi sent to route | hi_devices.maxpat

You can obtain data from the other buttons by following this same approach and appending their index values to the *route* arguments. Joysticks are a little different since they send out continuous values, unlike buttons that send out "on" or "off" values. Open the Max window and move the joystick to reveal the index of the joystick, temporarily disregarding the second number in the pair, the element.

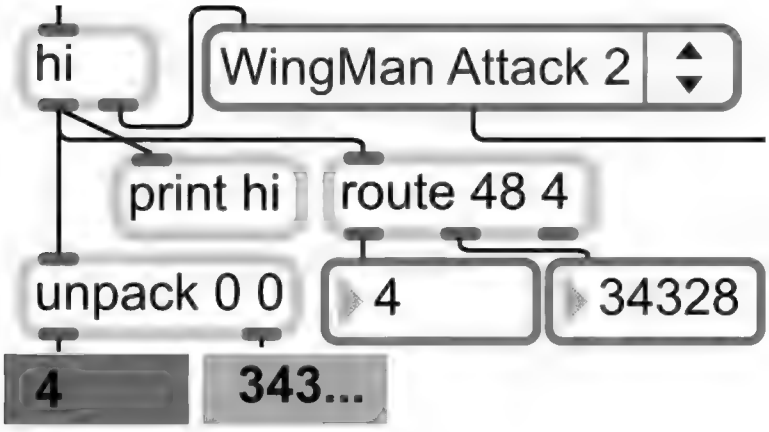


FIGURE 8.17

route object is given more indexes as arguments by which to obtain data | hi_devices.maxpat

For my device, the joystick index is *4*, so I can add this index value as an additional argument for *route*.

9. Double click the *route* object and add the address value of the device component—in this case, the joystick—you want to get data from
10. Create a *number* box
11. Connect the second outlet of *route* to the inlet of the newly created *number* box

After you've been able to index each button, knob, or joystick on your control device by adding the index value to *route*, it's time to get the element data into a format that's usable for your patch. Of course, herein lies the big question: how do you want to use this device in your patch?

There is certainly novelty in controlling scales with a joystick or chords with a button, and your students and colleagues will likely rank you high on the "oooh wow" factor for getting your dusty old joystick to play Rachmaninoff. More important, there should be some pedagogical way of using the control device to do whatever it is that your patch does. Does it make sense, for your purpose, to map the numbers sent by the joystick to the notes of a scale? Would these numbers be better mapped to control velocity? Should the buttons on the device cause chord changes? Should the buttons change timbres? It's really cool to use a videogame controller, but, apart from its cultural importance, does it help performance in such a way that makes it actually worth using? Is its cultural importance enough to justify using it for performance and composition activities in the population it will be presented to? These are all things to consider during this stage where, as I said before, mapping is everything.

Since you are already able to obtain numbers from the device, you can use a *toggle* for any two-state buttons like the button we first mapped. You can even use a *select* with the arguments *0* and *128* if desired.

For joystick-like controls where the data span a large range, determine the minimum and maximum values and use the *scale* object to map the data to a more useful set of numbers, like 0 127. In the case of the joystick I'm using, the minimum value is *0* and the maximum value is *65535* which I will map to the MIDI range *0–127*.

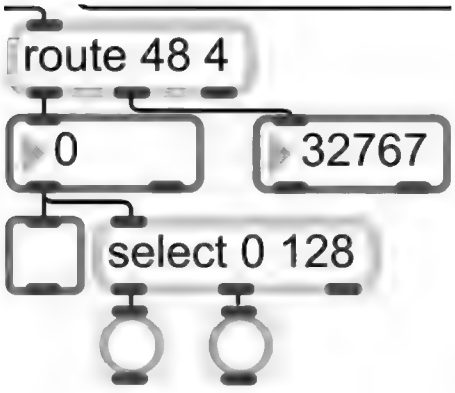


FIGURE 8.18
data from hi sent to select | hi_devices.maxpat

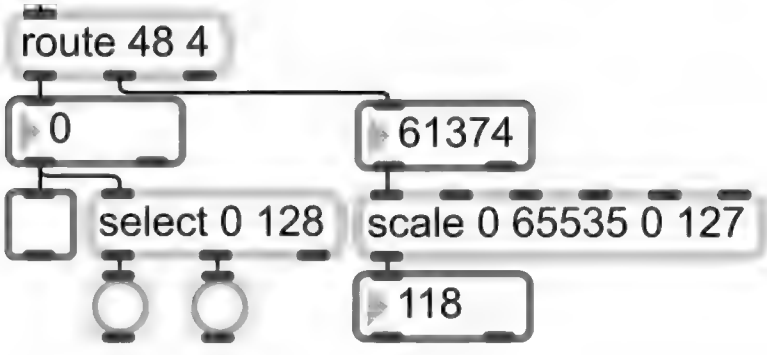


FIGURE 8.19
data from hi is scaled to a desired output range | hi_devices.maxpat

Copy the contents of this Help file into a new patch, and save the new patch as *hi_devices.maxpat*. Close the *hi_devices* patch. Close the Help file without saving the changes.

New Objects Learned:

- *switch*
- *mousestate*
- *scale*
- *screensize*
- *change*
- *hi*

Remember:

- To save patch settings using *pattr*, create an object called *pattr* and give it a name as an argument. Create an object called *autopattr* and give it the same name as an argument. Open the *Inspector* for any object whose settings you want to include in the save, and enter the same argument name as its *Scripting Name*. Piece of cake!

- The characters *#1* and *#2* in a *message* box are placeholders for arguments sent to the patch.

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Mouse Drawing—Introduction to drawing
  - Random Drawing—Working with random numbers
  - Procedural Drawing—Creating procedural code
  - Data Viewing—Visualizing data streams
  - Data Scaling—Mapping and scaling numerical information
  - Human-Interface Devices—Working with game controllers

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Imagine that you are working with special needs groups or individuals who have limited finger dexterity; the space bar is a big button that they can press. Design a musical instrument in Max that causes something musical (a chord, a single note, many chords, random diatonic chords, random tonal pitch generators, etc.) to happen when the user hits the space bar. In addition, use the mouse in this patch in some novel way. You may explore objects in the Modal Object Library and the EAMIR SDK to help in the creation of this instrument if desired.



Tools for Music Theory Concepts

In this chapter, we will design some tools to aid in the discussions of concepts related to music theory. In particular, we will discuss chord progressions, scale analysis, chord analysis, mode relationships, harmonic direction of chords, and harmonization. By the end of this chapter, you will have an arsenal of tools for explaining theoretical concepts of music.

Sometimes, theoretical concepts in music can be difficult to grapple with, even for professional musicians. As we discuss some different ways to address these concepts through software, try to think of demonstrating the theory concept as the goal, and the Max part of it as the means of reaching the goal. This will help you to program with the goal in mind and will help the way we reach that goal, through Max, to make more logical sense. Begin with the goal in mind!

Chord Progressions

Let's quickly build a patch that allows us to play back chords. As you'll recall, we used a patch like this in the Chapter 7: Example 2 in the EAMIR SDK. Let's open that patch. You may also build a chord patch from scratch if you prefer.

1. Click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*

2. In the *umenu* labeled *Examples*, click the second item *2.EAMIR _Chord_Basics.maxpat*
3. Click *File>Save As* and save the file as *chord_progressions.maxpat*

Suppose you wanted to discuss the chord progressions used in your favorite popular music song. Let's pretend that the chord progression is 1, 5, 6, 4 (I V vi IV) in the key of C Major (C, G, A minor, F). We can allow the user to play through each chord in the progression by entering these chords into a *coll*. Unlock the patch and

4. Create a new object called *coll* above the top *bpatcher* (you may have to move the entire patch contents down to make some room)
5. Create 2 *number* boxes above and below the *coll,* respectively
6. Connect the first outlet of the *number* box above *coll* to the inlet of *coll*
7. Connect the first outlet of *coll* to the inlet of the *number* box beneath *coll*

Lock your patch and

8. Double click the *coll* object to open its window and enter the following data:

```
1, 1;
2, 5;
3, 6;
4, 4;
```

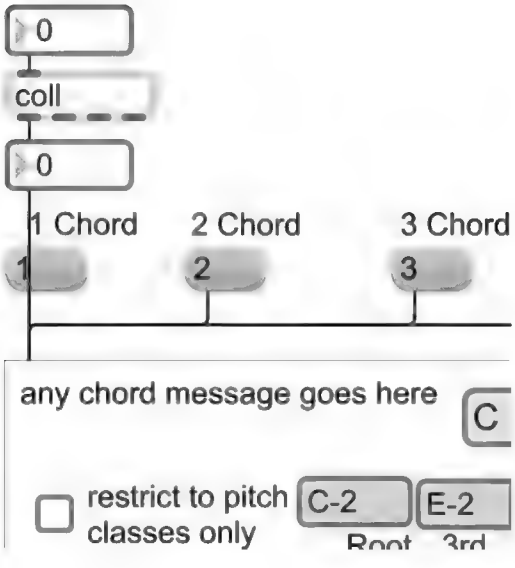


FIGURE 9.1

coll sends chord numbers to bpatcher | chord _progressions.maxpat

As you already know from our previous work with *coll*, if we send the number *1* to *coll*, it will look at index 1 and output the element at that address which, in this case, is also *1*. If we output the number *2*, *coll* will output the number *5*. We can then get each chord from our progression (1, 5, 6, 4) to play just by sending numbers to *coll* starting at *1* and increasing to *4*. The *counter* object would be perfect for this task as it is an object that simply counts within a range of specified numbers.

The *counter* object can take a minimum and maximum counting value as its first two arguments.[1] If we give *counter* the arguments *1* and *4*, it will count through each number beginning with 1 and ending with 4 each time it receives a bang. Unlock your patch and

9. Create a new object called *counter* with the arguments *1* and *4* above the *number* box connected to *coll*'s inlet
10. Create a *button*
11. Connect the outlet of the *button* to the first inlet of *counter 1 4*
12. Connect the first outlet of *counter 1 4* to the inlet of the *number* box *number* box connected to *coll*'s inlet

Lock your patch and click on the *button* connected to *counter 1 4*

Notice that after it reaches its maximum counting value, it goes back to the first value. A patch like this would allow you to demonstrate the same chord progression in a number of keys. If you set the *button* to receive a bang every time the space bar was pressed (*key*, *select 32*), you could use this patch as an instrument to play entire songs.

Think about all the popular songs that use only 4 chords. You can use a patch like this to play through those chords while singing. Though pop music is often repetitive and simplistic, it seems the ideal type of music to use in instances where repetition is necessary for pedagogical purposes.

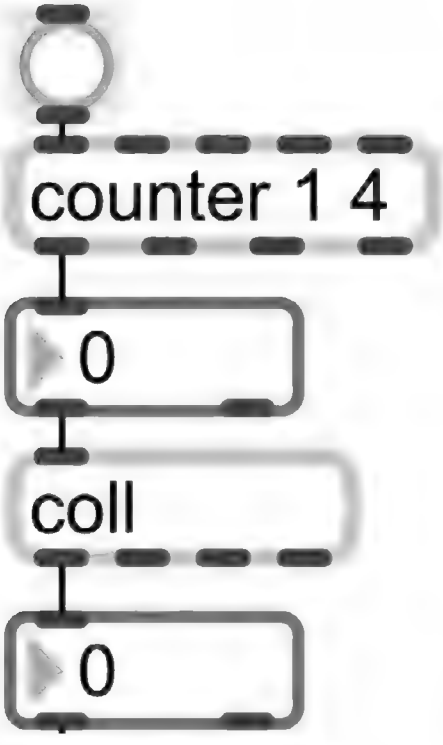


FIGURE 9.2

counter reports elements at first 4 indexes inside of coll | chord_progressions .maxpat

Save and close this patch. A shortcut abstraction version of what we just created is available in the Modal Object Library as an object called *modal_prog*.

13. Click on *Extras>Modal_Object_Library* from the top menu to view the main menu of the Modal Object Library
14. Click on the *message* box containing the text *modal_prog* to open the Help file for this object

1. An extra argument can be used to specify the counting direction, but we will not discuss that here.

This abstraction plays chord progressions in basically the same way that we just programmed. It takes the chord progression list as a *message* and formats each chord to an index within the *coll* located inside the abstraction. It's just another shortcut which you can use if desired. Imagine making an application that allows an individual to practice improvising over a bunch of different 4-bar progressions randomly chosen from the software. Close this Help patch.

Scale Analysis

If you can determine 7 discrete notes being used in a passage of music you can probably get close to naming the scale. Being able to determine the tonic with your ears is a big help. If you've ever dealt with harmonic analysis software before, you've likely been less than impressed with the results. The reality is that there are often many contextual matters that make analysis less "cut and dried" than you might hope. However, we will demonstrate the concept of scale analysis by building a simple scale analyzer in Max. From this patch, you can go on to make similar patches to recognize chords and other harmonic patterns and structures.

When I first formally learned about analysis, chords, or scales, my teacher told me to "put all of the notes in a hat" so I knew what I was looking at. His suggestion concerned filtering out repeated notes and notes at different octaves, and making a reduction of the pitch material so that it fit into whatever mental constructs I had about tonality. Not a bad way to think about analysis. The difficulty, of course, comes in giving the computer a "mental construct" of music. Since we have already done some work with the C Major scale, we'll start off with this scale and see if we can write a program to analyze the scale correctly.

The first thing we'll need is some object that will take all of the MIDI notes we play "into a hat." No, there's no "hat" object, but we can use an object called *bag*. The *bag* object will take all of these MIDI notes, filter out duplicates, and send them out when it receives a bang.

1. Create a new patch
2. Create a new object called *notein*
3. Create a new object called *bag*
4. Connect the first outlet of *notein* to the first inlet of *bag*

The pitches from *notein* will be stored in *bag* when *bag* first receives the number *1* in its second inlet. For this reason, it's necessary for us to send *bag* the number *1* when the patch loads, so that everything it receives will be stored.

1. Create a new object called *loadmess* with the argument *1*
2. Connect the outlet of *loadmess 1* to the last inlet of *bag*
3. Lock your patch and double click the *loadmess 1* object to send the number *1* to *bag*'s second inlet

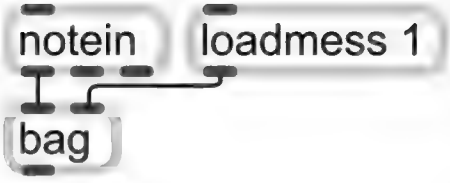


FIGURE 9.3

MIDI notes are collected in bag | scale_analysis .maxpat

Let's connect the outlet of *bag* to the inlet of a *print* object so we can see if it's working. We'll also need a *button* to trigger the release of *bag*'s contents as well as a way to clear the notes in *bag* after we analyze them so that they don't accidentally make their way into the next scale we try to analyze. The *bag* object will receive the message *clear* to perform this action, so we can use *trigger* to send a bang to *bag* to output the pitches followed by the message *clear* to remove the pitches in the *bag*. Unlock your patch and

4. Create a new object called *print* beneath *bag*
5. Connect the outlet of *bag* to the inlet of *print*
6. Create a new object called *trigger* with the arguments *clear* and *b*
7. Connect both outlets of *trigger* to the first inlet of *bag*
8. Create a *button* above *trigger*
9. Connect the outlet of *button* to the inlet of *trigger*

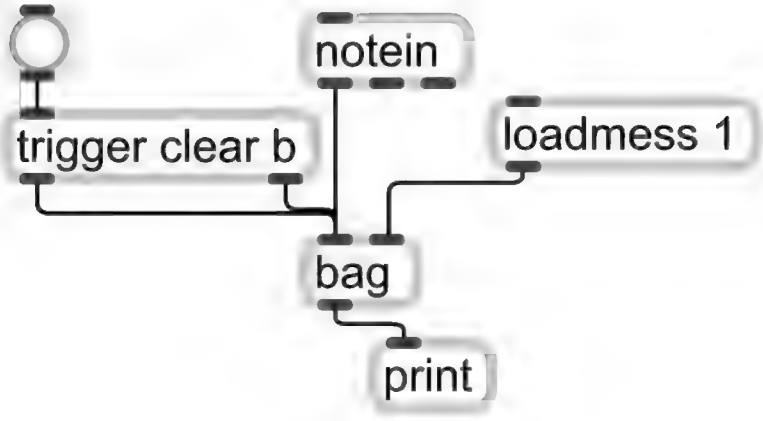


FIGURE 9.4

trigger bangs collected MIDI pitches from bag to print then clears the bag | scale_analysis.maxpat

Lock your patch and

10. Play a one-octave C Major scale starting on middle C
11. After playing the scale, click the *button* and open the Max window to reveal the notes that were emptied from *bag*

Notice that the Max window shows each separate value outputted from *bag* because they don't leave *bag* in one list; each number is outputted separately.[2] What we need is a way to get all of these notes into one list so that we can analyze the distance between each note in order to determine what scale it is.

2. There is an object called *iter* that takes a list of numbers and also outputs each one separately, but what we need for this patch is the opposite: take a series of separately outputted notes and group them into one list.

(Remember our "whole step/half step" activity in Chapter 4?) The *thresh* object will take a bunch of numbers received within a certain threshold of time, default 5 milliseconds, and group those numbers into a list. Perfect! Unlock your patch and

12. Create a new object called *thresh*
13. Connect the outlet of *bag* to the first inlet of *thresh*
14. Disconnect the patch cord connecting the outlet of *bag* to the inlet of *print* and, instead, connect the outlet of *thresh* to the inlet of *print*

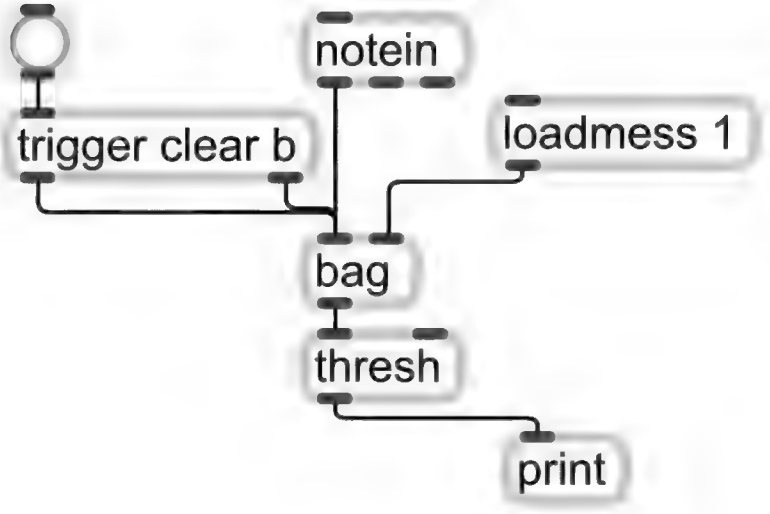


FIGURE 9.5

trigger bangs the contents of bag to thresh where they're made into a list | scale_analysis.maxpat

Lock your patch and

15. Play a one-octave C Major scale starting on middle C
16. After playing the scale, click the *button* and open the Max window to reveal the notes in a single list that were emptied from *thresh*

Now our patch is heating up! Let's put a little protection into our patch to ensure that all of the notes played are in some sort of order. This is particularly helpful if, instead of playing an ordered scale like we've been playing, you were playing notes from chords within the context of a passage of music. You may play all of the notes of the C Major scale with C as your lowest tone, but starting with the notes in this order: D, F, A, E, G, B, C. You would have difficulty analyzing the distance between those notes because they aren't in order.

Luckily, since our notes are in a list, we can use one of the many list processing objects in Max. One object called *zl* processes lists in a number of different modes depending on the argument you give it. For example, *zl sort* will sort a list in ascending order, while *zl rev* will reverse a list. There are many different

modes of *zl*. The mode that will work best for this application is *zl sort*. Unlock your patch and

17. Create a new object called *zl* with the argument *sort*
18. Connect the outlet of *thresh* to the first inlet of *zl sort*
19. Disconnect the patch cord connecting the outlet of *thresh* to the inlet of *print* and, instead, connect the outlet of *zl sort* to the inlet of *print*

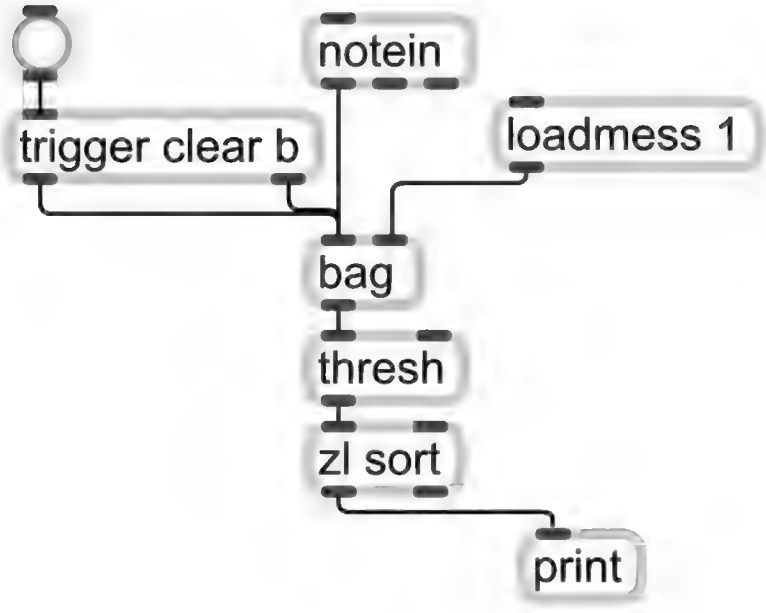


FIGURE 9.6
pitches in the list made by thresh are sorted into numerical order by zl sort | scale_analysis.maxpat

Lock your patch and

20. Play a one-octave C Major scale starting on middle C
21. After playing the scale, click the *button* and open the Max window to reveal the notes in a single list that were emptied from *thresh* and ordered by *zl sort*

We now have an ordered list of notes for which we can easily identify the number of semitones between each pitch. In fact, we did just the opposite task in Chapter 4 when we built a scale by adding numbers according to a whole step/half step pattern. The task at hand is to *unpack* each number from our list and subtract one scale note from the one preceding it to determine the distance between the two pitches in semitones.

We will write a mathematical expression with *expr $i2 – $i1* to do the necessary math to determine the distance between each pitch. We can use other objects to do this math, but it doesn't hurt to get some more practice using *expr*. What we'll be left with is a series of whole steps and half steps for the scale we played where *2* is equal to a whole step (2 semitones) and *1* is equal to a half step (1 semitone); for example: *2 2 1 2 2 2 1* for a major scale. We will then have to

get those whole steps and half steps into a list using *pack* so that we can compare the pattern to the pattern of scales we know. Unlock your patch and

22. Create a new object called *unpack* with 8 zeros as arguments: *0 0 0 0 0 0 0 0*
23. Connect the outlet of *zl sort* to the inlet of *unpack*
24. Create 7 new objects called *expr* organized horizontally with the following math expression as an argument: *$i2 – $i1*
25. Connect the first outlet of *unpack* to the first inlet of the first *expr* object
26. Connect the second outlet of *unpack* to the second inlet of the first *expr* object and the first inlet of the second *expr* object
27. Connect the third outlet of *unpack* to the second inlet of the second *expr* object and the first inlet of the third *expr* object
28. Connect the fourth outlet of *unpack* to the second inlet of the third *expr* object and the first inlet of the fourth *expr* object
29. Connect the fifth outlet of *unpack* to the second inlet of the fourth *expr* object and the first inlet of the fifth *expr* object
30. Connect the sixth outlet of *unpack* to the second inlet of the fifth *expr* object and the first inlet of the sixth *expr* object
31. Connect the seventh outlet of *unpack* to the second inlet of the sixth *expr* object and the first inlet of the seventh *expr* object
32. Connect the eighth outlet of *unpack* to the second inlet of the seventh *expr* object
33. Create 7 *number* boxes placing one beneath each *expr* object
34. Connect the outlet of each of the *expr* objects to the inlet of the *number* box beneath it
35. Create a new object called *pack* with the arguments *0 0 0 0 0 0 0*
36. Connect the first outlet of each *number* box to the inlets of *pack* so that the *number* box on the far left is connected to *pack*'s first inlet
37. Disconnect the patch cord connecting the outlet of *zl sort* to the inlet of *print* and, instead, connect the outlet of *pack* to the inlet of *print*

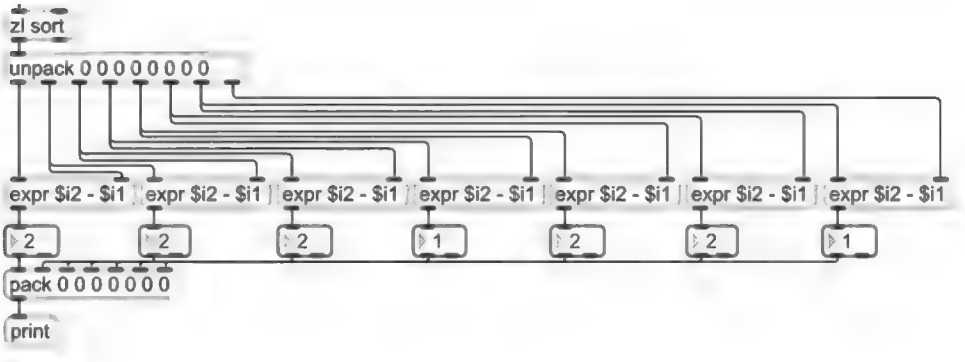


FIGURE 9.7

intervals between notes in the sorted list are unpacked and determines through subtraction | scale_analysis.maxpat

Lock your patch and

38. Play a one-octave C Major scale starting on middle C
39. After playing the scale, click the *button* and open the Max window to reveal the whole step/half step pattern for the major scale

Congrats! You're basically done at this point. Now, you simply need some sort of database that understands that a list like *2 2 1 2 2 2 1* means "Major Scale." You can create your own using a series of *match* objects like the abstraction we looked at in Chapter 6. You could also use the *modal_change* external object to identify these patterns since the object already has these step patterns built into it. You'd simply need to connect the outlet of *pack* to the second inlet of *modal_change*. Of course, we'll need to somehow determine the tonic for this scale.

As I mentioned before, it can be difficult to determine the tonic in the context of music that is constantly in motion. For our purposes, now, we will say that the lowest note played was the tonic. This will allow us to retain modal context when analyzing. For example, the notes G A B C D E F where G is the lowest note, will be interpreted as G Mixolydian, even though it has the same notes as C Major (Ionian).

Since *zl sort* organized our notes in ascending order, we can get the lowest note from the first outlet of *unpack*. When we get this note from *unpack*, we'll reduce it to a pitch class with % *12* so that *modal_change* can work with it. Unlock your patch and

40. Create a *number* box
41. Connect the first outlet of *unpack* to the inlet of the newly created *number* box
42. Create a new object called % with the argument *12* beneath the *number* box
43. Connect the first outlet of the *number* box to the first inlet of % *12*
44. Create a new object called *modal_change*
45. Connect the outlet of % *12* to the first inlet of *modal_change*
46. Connect the outlet of *pack* to the second inlet of *modal_change*
47. Delete the *print* object

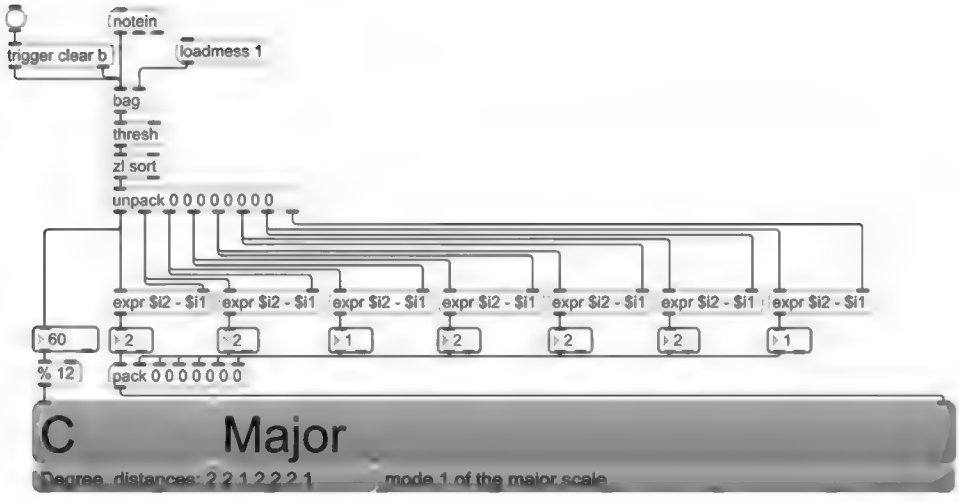


FIGURE 9.8

intervals are packed into a list by which we can determine a mode based on the pattern | scale _analysis.maxpat

Lock your patch and

48. Play a one-octave C Major scale starting on middle C
49. After playing the scale, click the *button* and notice the display of *modal_change*
50. Play a one-octave C Lydian mode (just like C Major, but play F♯ instead of F natural)
51. After playing the scale, click the *button* and notice the display of *modal_change*
52. Play a one-octave C Lydian b7 mode (just like C Major, but play F♯ instead of F natural and B♭ instead of B natural)
53. After playing the scale, click the *button* and notice the display of *modal_change*

Imagine how a discussion of scales and modes can be fostered using a patch like this. In my experience, explaining these concepts with the use of an interactive visual aid is much easier than leaving everything abstract and telling the student to simply "visualize the keyboard in your mind." Save this patch as *scale_analysis .maxpat* and close the patch.

The abstraction *modal_analysis+* in the Modal Object Library is similar to the type of patch we just made. Another abstraction called *modal_chord _analysis* shows the similarities in design between this patch and the basics of analyzing root position triads. However, instead of using those abstractions, I encourage you to make your own abstraction from this patch. This can be done by replacing the *notein* object in this patch with an *inlet* object, connecting an *inlet* to the *button*, and replacing the *modal_change* object with two *outlet* objects (receiving from *modal_change*'s two inlets). You will then be able to use this patch as an abstraction inside of a parent patch. Open the file *My _Scale_Analysis.maxpat* from the *Ch 09 Examples* folder inside of the *patches* folder.

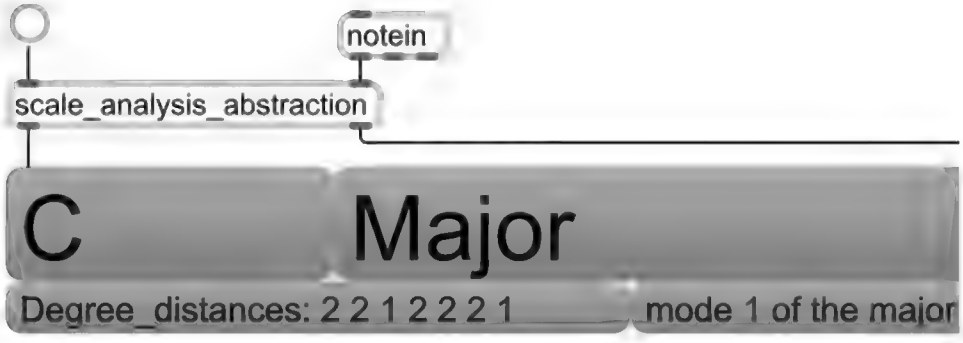


FIGURE 9.9

abstraction analyzes data received from notein and sends data to modal _change | My_Scale _Analysis.maxpat

Notice that the object contains a reference to the patch *scale_analysis_abstraction*, a modified version of the *scale_analysis* patch with *inlet*s and *outlet*s added.

54. Double click the *scale_analysis_abstraction* object to view its contents in a new window

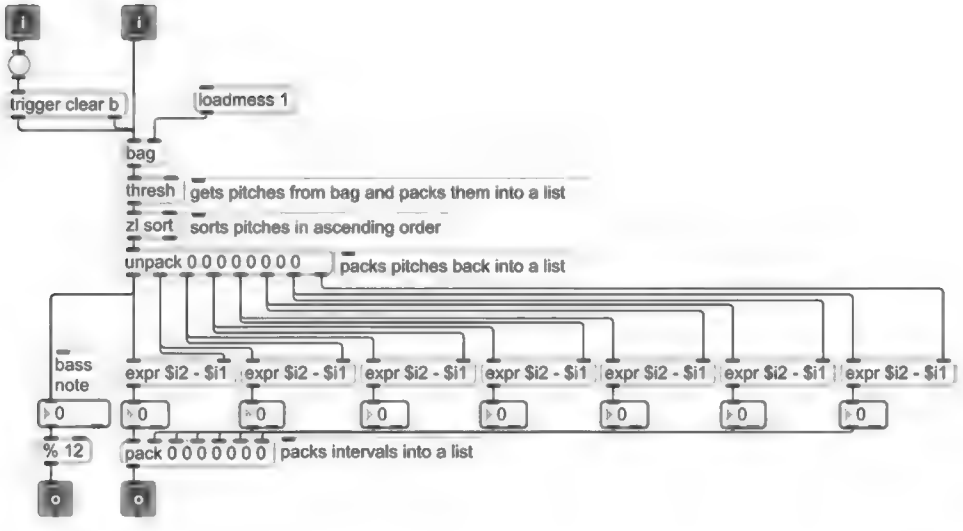


FIGURE 9.10

abstraction shows parts of the patch replaced by inlet and outlet objects | scale_analysis_abstraction .maxpat

As you can see, it's basically the same patch with some *inlet* and *outlet* objects. I've even added comments to each *inlet* and *outlet* through the Inspector. Close the abstraction window.

55. Click *File>Save As* and save the *My_Scale_Analysis* patch as *mode_relationships.maxpat* in the same folder as the original file. We'll be using the *mode_relationships* patch in the next exercise

Mode Relationships

In an explanation of modes and tonal centers, it is not uncommon to discuss the relationships between modes. For example, in the 7-note mode of C Major, there are 42 modes sharing 6 of the 7 scale degrees accessible by raising or lowering

one scale degree a semitone. The mode of C Major (C D E F G A B) can become C Lydian (C D E F♯ G A B) by changing just one scale degree, scale degree 4. C major can also become C♯ Locrian b4 (C♯ D E F G A B), the seventh mode of the D melodic minor scale, just by moving scale degree 1 up a semitone.

As mentioned, there are 42 modes related to major scales in this way. There are also 21 modes related to the harmonic minor scale, 18 modes related to the melodic minor scale, and 21 modes related to the harmonic major scale,[3] all accessible through common-tone modulation by increasing or decreasing a single scale degree by one semitone. Understanding this concept could allow individuals to explore these relationships in their own compositions as well as detect modulations when they are analyzing scores.

1. If it's not still open, open *mode_relationships.maxpat* (it should be in the same folder as the file *scale_analysis_abstraction .maxpat*)

Since we know that C Lydian is related to C Major in that they have 6 scale degrees in common, let's make a patch so that when a user to plays a major scale, the user will see that the Lydian mode is related to that major scale. Unlock your patch and

2. Disconnect both outlets of the *scale_analysis_abstraction* object from the inlets of *modal_change*
3. Create a new object called *match* with the arguments *2 2 1 2 2 2 1*
4. Connect the second outlet of the *scale_analysis_abstraction* object to the inlet of *match*
5. Create a *message* box containing the text *lydian*
6. Connect the outlet of *match* to the first inlet of the *message lydian*
7. Connect the outlet of the *message lydian* to the second inlet of *modal_change*

As you'll recall, the second outlet of the *scale_analysis_abstraction* object outputs the step pattern for the scale played by the user. When a user plays a major scale, the list *2 2 1 2 2 2 1* will be sent from the abstraction and received by the *match* object. The *match* object is like *select* in that it looks at data coming in its inlet and sends a bang if the input matches one of its arguments. Unlike *select*, however, for *match* to send out a bang, the input must be exactly the same as the arguments in *match*. Since the *match* in our patch has the same exact arguments as the message it's receiving, *match* will send out a bang to the *message* box *lydian* which will be received by *modal_change*. However, we still need to send

3. The term Harmonic Major is used to describe a major scale with a flatted sixth scale degree such as C D E F G A♭ B C.

a tonic to *modal_change*'s first inlet. In this illustration, the tonic, C, is the same for C Major and C Lydian. We will, however, build our patch so that we can accommodate related modes with different tonics.

8. Create an object called *int*
9. Connect the first outlet of the *scale_analysis_abstraction* object to the second inlet of *int*
10. Create a *button* above the *int*
11. Connect the outlet of *match* to the inlet of the *button*
12. Connect the outlet of the *button* to the first inlet of *int*
13. Create a new object called + with the argument *0*
14. Connect the outlet of *int* to the first inlet of + *0*
15. Create a *number* box beneath + *0*
16. Connect the outlet of + *0* to the inlet of the *number* box
17. Connect the first outlet of the *number* box to the first inlet of *modal_change*

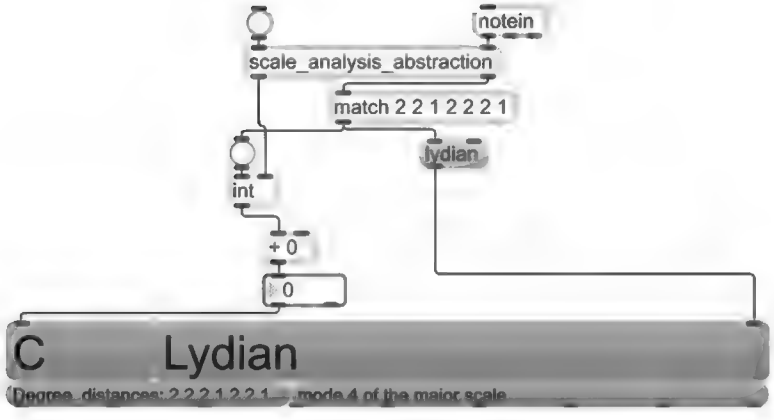


FIGURE 9.11

match detecting the major scale pattern and triggering a related mode | mode_relationships .maxpat

Lock your patch and

18. Play a one-octave C Major scale
19. After playing the scale, click the *button* and notice that the *match* object triggers *modal_change* to show the related mode *C Lydian*

As you can see, when the scale is matched, *match* sends a bang to both the *message lydian*, which *modal_change* understands, and a bang to the *int* that was storing the tonic sent from the abstraction. The tonic then has 0 added to it and is sent to the first inlet of *modal_change*.

Suppose you wanted to use the related mode of *C♯ Locrian b4* instead of *C Lydian*. There are only a few things that need to change: the *message* box containing the text *lydian*, and the amount you add to the + object to offset the

tonic. For *C♯ Locrian b4*, the tonic of a major scale should have *1* added to it. Unlock your patch and

20. Double click the *message* box and change its contents from *lydian* to *locrian_b4*
21. Double click the *+ 0* object and change its argument from *0* to *1*

Lock your patch and

22. Play a one-octave C Major scale
23. After playing the scale, click the *button* and notice that the *match* object triggers *modal_change* to show the related mode. In this case, it will have an enharmonic spelling of the tonic

If you are able to list all the related modes, or even make up your own types of modal relationships, you can populate the different choices into a *umenu*. Several abstractions like this exist in the Modal Object Library including *modal_shift*, *modal_shiftlist*, *modal_mediant*, and *modal_mutation*. All these abstractions operate just like the patch you just made except that they already have the relationships coded into them. Perhaps there are some types of mode relationships that you'd like to demonstrate visually to a class; now you can write a patch to show those relationship. Save and close this patch.

Harmonic Direction

Let's look at an existing patch in the EAMIR SDK made for use with interactive white boards and touchscreen computers.

1. Click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
2. In the *umenu* labeled *Examples*, click the item labeled *EAMIR _Smart_IWB.maxpat*

When a user clicks on a purple square, a chord will sound and the square will turn dark purple. At the same time, a blinking red rectangle will appear around all of the chords that the selected chord tends to resolve to. For instance, click on the purple button with the *5* beneath it. It will play the *5* chord in the selected key, and red blinking rectangles around the squares labeled *1*, *6*, and *8* begin to blink showing that *5* chords tend to resolve to the tonic chord or the submediant chord. Clicking the purple square again will release the chord.

This patch is currently in Presentation mode. Unlock the patch and put it in Patching Mode. You will see a number of purple squares in rows with *umenu* objects beneath them. These purple squares are actually *bpatcher*s showing a file called *EAMIR_button.maxpat*.

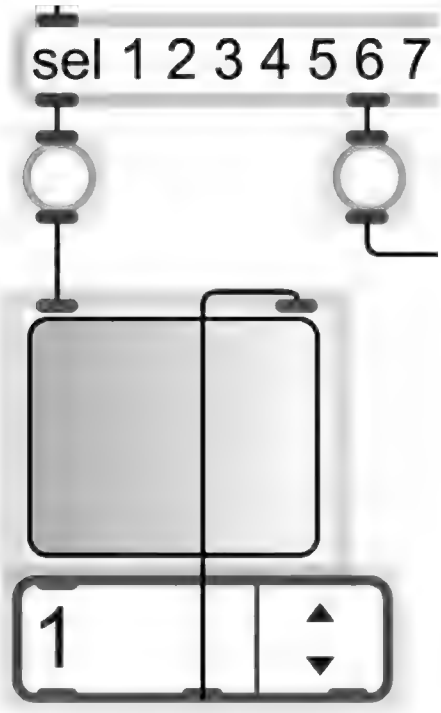


FIGURE 9.12
small bpatcher | EAMIR _Smart_IWB.maxpat

3. Ctrl+click (Mac) or right click (Windows) one of these *bpatchers* and select *Object>Open Original "EAMIR_button.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching Mode.

What you see inside the patch is not very complicated: a couple of objects called *ubutton* set to change color when they receive a bang.

4. Turn on the *metro 200* to see one of the *ubutton* objects blink red
5. Click the other *ubutton* to see it turn purple

The *ubuttons* are great objects for placing over images (placed with *fpic*) or, in general, as a nice alternative to using the *button* object. Notice that the purple square is just an object called *panel*—an object used for patch design and decoration that shows a color or a several colors. The *ubutton* goes on top of the *panel* in Presentation mode, so that when a user clicks on the *panel*, the *ubutton* receives the click. Close the *EAMIR_button* window.

In this patch, I've used the purple *ubutton* to show when a user has clicked on the purple square to play a chord, and a red *ubutton* to show the common harmonic direction of that chord. The *metro* inside of the *EAMIR_button*, as you know, makes the *ubutton* turn on and off at 200 milliseconds creating the illusion that it is blinking. So in this patch, all that is really necessary to show the harmonic direction of chords is the knowledge of harmonic direction and a way

to turn the *metro* objects on and off for certain chords thus causing the *ubutton* to blink.

Lock the patch and double click the *coll* object with the name *eamir_smart_iwb_chords*. This reveals a list of data in which the index is a chord name and the elements are chord functions that we want to start blinking. When the fifth *ubutton/bpatcher* is pressed, the chord name is sent to the *coll* which receives the input *5* as an index value and outputs the numbers *1*, *6*, and *8* to the *select* object. The *select* object sends out a bang to the *bpatcher*s causing their *metro* objects to turn on, which causes the red blinking.

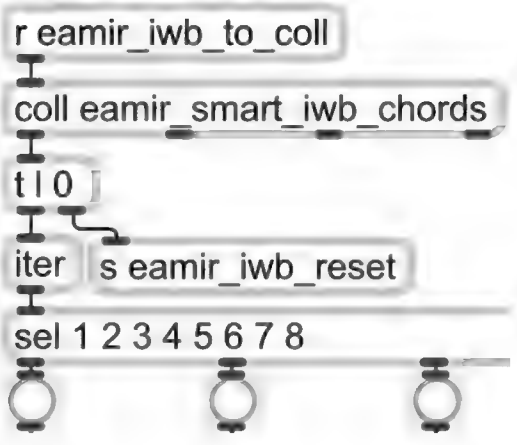


FIGURE 9.13
data from coll causes some bpatchers to blink | EAMIR_Smart_IWB .maxpat

The rest of the patch is like the *Chord Basics* patch we looked at in Chapter 7. In fact, the *ubutton/bpatcher* is just another control interface for a patch that functions like other patches we've looked at. Of course, this patch has the added feature of providing some guidance for users regarding the way we traditionally think about harmonic direction tendencies for chord resolution. Close this patch and close the *EAMIR SDK* menu.

Harmonization

Once we have established a scale, we can obtain a good understanding of the types of chords you might want to use to harmonize that scale. In a major scale, scale degree 1 is present in the I chord, the IV chord, and the vi chord. As a result, you can use an object like *select* in a patch to look for C notes sent from *notein* so that when *select* sees a C, it will bang the chord of your choosing to harmonize the note.

1. Click on *Extras> Modal_Object_Library* from the top menu to view the main menu of the *Modal Object Library*
2. Click the *message* box containing the text *modal_pc_match* to open up the Help file for the selected abstraction

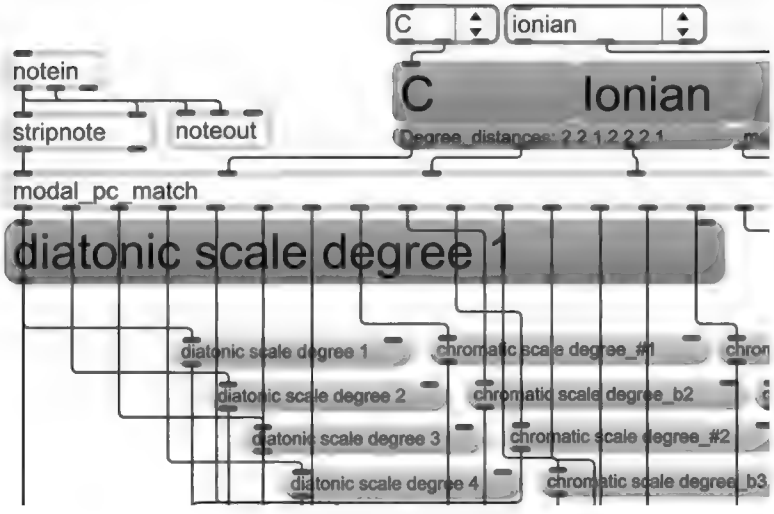


FIGURE 9.14

modal_pc_match abstraction detects a scale degree related to the selected mode | modal_pc_match.maxhelp

The *modal_pc_match* abstraction does just what we described earlier. When the Help file is loaded, the notes of the C Major scale are sent from *modal_change* to the inlets of *modal_pc_match*. Each note of the scale is stored in a *match* object so that when a note is played on a MIDI keyboard, the note will pass through each *match* object, reporting when it finds a match. The abstraction has outlets for each diatonic and chromatic scale degree. Depending on which scale degree the incoming note matched, the abstraction will send a bang from one of its outlets. The abstraction also accounts for chromatic notes and has *match* objects for those as well. If you think about it, this abstraction is a fairly simple concept.

3. Play some notes in the key of C Major and notice that the patch reports the scale degree of the note played. Play some chromatic notes as well.
4. Change the key and observe how the note C is scale degree 1 in the key of C Major and scale degree 4 in the key of G Major and so on.

Keeping scales and scale degrees organized with the ability to show clear connections between tonal centers can be a huge help in explaining concepts of modulation, chromaticism, and other harmonic aspects of music including the function of specific pitches with respect to varying tonal centers and modes. We can also use the information to make generalities about harmonizing scale degrees with chords.

For example, if you would like to harmonize a particular scale degree with a chord, you can simply connect the appropriate outlet of *modal_pc_match* to the chord *message* of your choosing. For an illustration of this,

5. Open the file *pc_match_chordmaker.maxpat* in the *Ch 09 Examples* folder

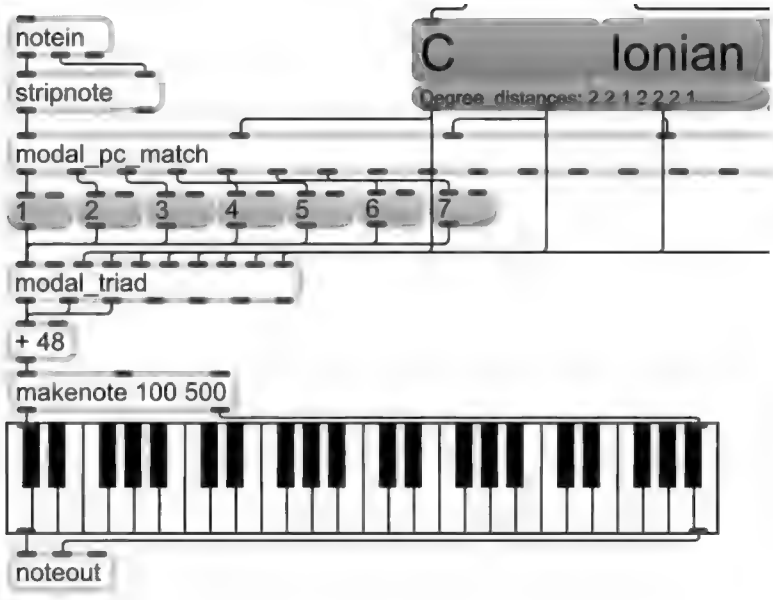


FIGURE 9.15

abstraction bangs message boxes with chord numbers | pc _match_chordmaker .maxpat

In this patch, *message* boxes containing chord functions are connected to the *modal_triad* object. When the *match* objects in *modal_pc_match* detect a match, a bang is sent to one of these chord *messages.*[4] The *stripnote* object seen here simply "strips" the "note off" message from sounding so that notes are not harmonized twice: once when the key is pressed down, and once when the key is released. Close the *pc_match_chordmaker.maxpat* patch and the *modal_pc _match* Help file.

A similar abstraction called *modal_fuzz_harm* uses the same concept to harmonize chords using "fuzzy logic." That is, a number of chords that a scale degree can be harmonized with are stored in a database, and when a bang is sent, one of the chords is chosen based on probability weights defined by the user. That may sound more complicated than it actually is, so let's open up the *modal_fuzzharm* abstraction's Help file and take a look at the concept.

6. If it's not still open, click on *Extras>Modal_Object_Library* from the top menu to view the main menu of the *Modal Object Library*
7. Click the *message* box containing the text *modal_fuzzharm* to open up the Help file for the selected abstraction

As you can see, the outlets of *modal_pc_match* are connected to instances of *modal_fuzzharm.*

8. Double click the first instance of *modal_fuzzharm* to view its contents

4. Note that the chromatic chord outlets do not have a chord *message* connected to them, so they will pass right through without being harmonized until we supply them with a chord (secondary dominant chords or other tonicizations would work nicely here).

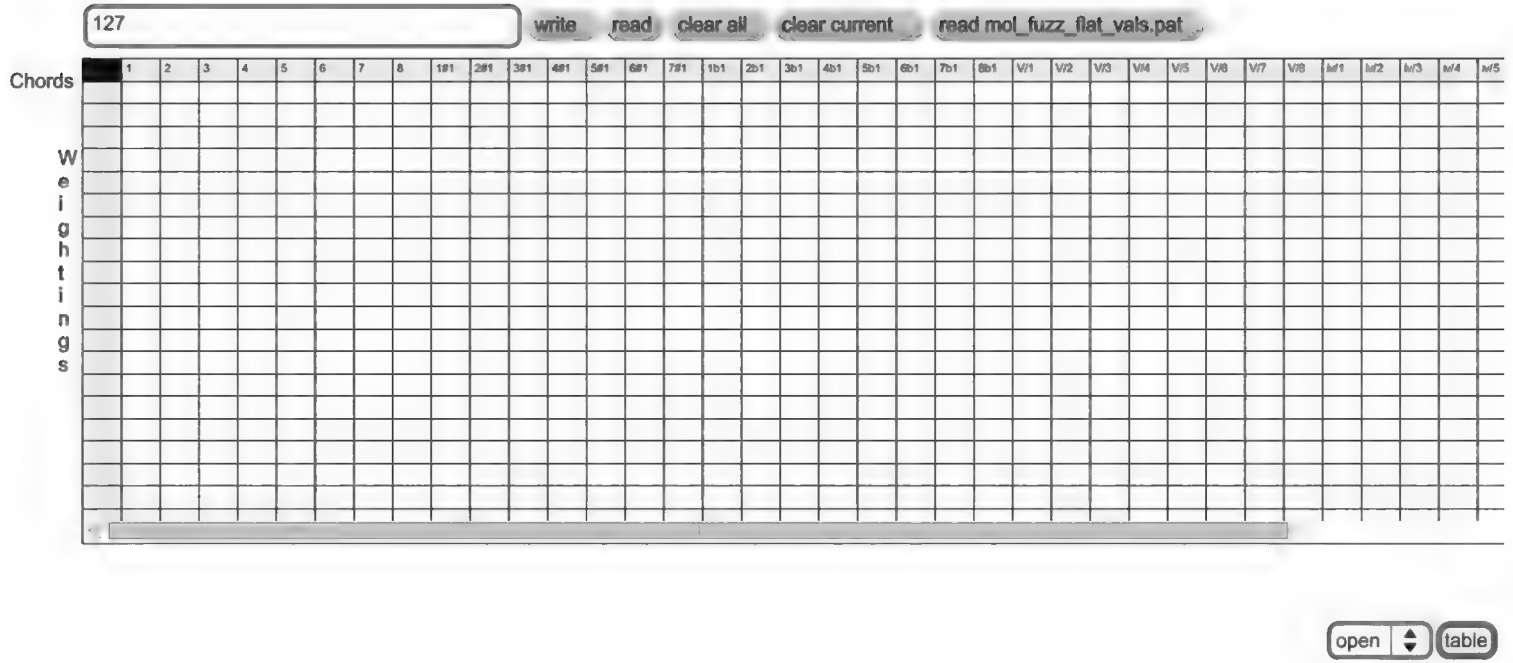

127
write
read
clear all
clear current
read mol_fuzz_flat_vals.pat
Chords
Weightings
1 2 3 4 5 6 7 8 1#1 2#1 3#1 4#1 5#1 6#1 7#1 1b1 2b1 3b1 4b1 5b1 6b1 7b1 8b1 V/1 V/2 V/3 V/4 V/5 V/6 V/7 V/8 iv/1 iv/2 iv/3 iv/4 iv/5
open
table

FIGURE 9.16

spreadsheet of chords within modal_fuzzharm abstraction | modal _fuzzharm.maxhelp

The large spreadsheet-looking object is an object called *jit.cellblock*. For the purpose of this patch, think of it as a spreadsheet with a bunch of chords at the top of each column. When a user clicks on a cell in *jit.cellblock*, it increases the value of a corresponding point in the *table* beneath the *jit.cellblock*.

9. Double click the *table* object or select "open" from the *umenu* at its left

Notice that there are 3 points on this *table*. These 3 points correspond to the chord functions *1*, *4*, and *6* as they appear in *jit.cellblock*. What this means to us is that when a scale degree is matched by *modal_pc_match*, a bang will be sent to *modal_fuzzharm* which will send out one of these 3 chords.

As you'll recall from our discussion of *table* in Chapter 6, when a bang is sent to *table*, the indexes are sent to its outlet, not randomly, but based on the weighting of each element. In other words, the lower down on the *jit.cellblock* you click for one of the columns, the higher up the element will appear for each index in the *table*, increasing that chord's chance of being selected for harmonization when a bang is sent from *modal_pc_match*. It's a type of probabilistic harmonization.

10. Play some notes in the key of C Major and notice that the harmonization sometimes changes as a result of the weighted values in the *table* and their association with chord functions

Patches like this can be particularly useful if an individual is working with harmonizing a melody and would like to experiment with a number of different options. They are also interesting when used to harmonize monophonic instruments, especially when the software is given a number of interesting chords to "choose" from when harmonizing a note. Close this Help file as well as the *Modal_Object_Library* menu.

We have spent a great deal of time discussing harmony. In the next chapter we will discuss some aspects of rhythm, time, and sequencing.

New Objects Learned:

- *counter*
- *modal_prog* [abstraction]
- *bag*
- *iter*
- *thresh*
- *zl* (all modes)
- *modal_analysis+* & *modal_analysis* [abstraction]
- *modal_chord_analysis* [abstraction]
- *modal_shift* [abstraction]
- *modal_shiftlist* [abstraction]

- *modal_mediant* [abstraction]
- *modal_mutation* [abstraction]
- *ubutton*
- *panel*
- *modal_pc_match* [abstraction]
- *stripnote*
- *modal_fuzzharm* [abstraction]
- *jit.cellblock*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - Picture UI Elements—Designing user interfaces with custom image elements
  - Cellblock—Working with a visual spreadsheet interface

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that will help you explain modulation, or how scales relate to each other. You may use any of the objects or abstractions we've discussed. Keep in mind UI objects like *kslider* and *nslider* that will aid in your visualization of these concepts.



Working with Time

In this chapter, we will discuss aspects of time, rhythm, and the sequencing of events. When used in combination with what you know about generating pitch material, this chapter will help you to create interactive performance and composition systems, as well as create patches that demonstrate rhythmic complexity. By the end of this chapter, you will have created patches that can record and loop MIDI sequences as well as a number of patches that work with notes over time.

Sequencing

Digital Audio Workstations (DAWs) are complex programs that generally all allow users to sequence music in some way by recording MIDI and audio, and make edits to the recording data. Popular commercial DAWs include Apple Logic and GarageBand, Digidesign Pro Tools, Sony Acid, and Ableton Live, to name a few. The last thing you'd probably want to do in Max is write a patch that operates like one of these DAWs. In fact, one of the great things about Max is that since it's not a DAW but a programming language, it's so unlike traditional DAWs that it allows you to write any kind of program you want (like our synth that used the wheels to play pitches in Chapter 2).

However, there are some aspects of these DAWs that we will want to incorporate into our patches. In particular, recording MIDI and playing it back is made possible in Max through an object called *seq* (as in *seq*uencer).

1. Create a new patch
2. Create a new object called *seq*

The *seq* object can record and play back raw MIDI data. Since we're dealing with raw MIDI data as opposed to just pitches, we'll use the *midiin* and *midiout* objects. In fact, just to protect ourselves, let's also include the "raw MIDI" version of *flush*, *midiflush*, to avoid any stuck notes.

3. Create a new object called *midiin* above the *seq* object
4. Connect the outlet of *midiin* to the inlet of *seq*
5. Create a new object called *midiflush* beneath *seq*
6. Connect the outlet of *seq* to the inlet of *midiflush*
7. Create a *button* next to *midiflush*
8. Connect the outlet of the *button* to the inlet of *midiflush*
9. Create a new object called *midiout*
10. Connect the outlet of *midiflush* to the inlet of *midiout*

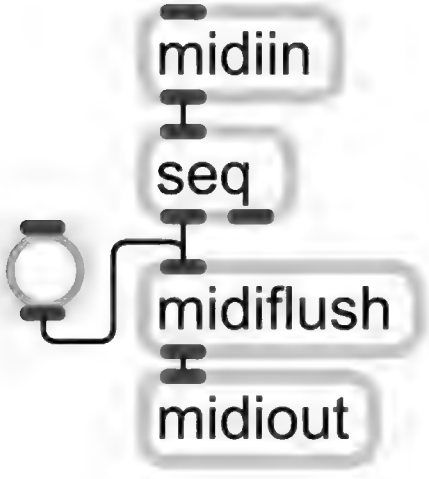


FIGURE 10.1
seq object used for sequencing MIDI data | MIDI_looper.maxpat

The *seq* object can take a number of *messages* to define the way it will function. For example, if you send it the *message record*, it will begin recording all MIDI data received from the *midiin* object. The *message stop* will stop the recording. The *message start* will play back what was recorded. In addition, the *message read* allows a user to load a single channel[1] MIDI file and play it back while the *message write* can save recorded MIDI data into a file. It's a really powerful object.

11. Create 3 new *message* boxes containing the text *record*, *start*, and *stop,* respectively
12. Connect the outlet of each *message* box to the inlet of *seq*
13. Connect the outlet of the *message stop* to the inlet of the *button* above *midiflush* to ensure that no notes are left "stuck" when the recording is stopped

1. A Type 1 MIDI file is one where MIDI data exist on multiple MIDI channels whereas a Type 0 MIDI file has all instrument channels merged to a single channel. *Seq* works with single channel, Type 0, MIDI files.

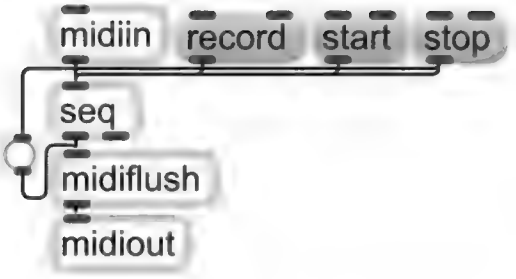


FIGURE 10.2
messages added to seq object | MIDI_looper .maxpat

Your patch is now equipped to record a performance from our MIDI keyboard and play it back. Lock the patch and

14. Click the *message record*
15. Begin playing something on your MIDI keyboard
16. Press the *message stop* when you have finished performing
17. Press the *message start* to play back your performance

Fantastic! Let's transform this patch into something a little bit different. The second outlet of *seq* sends a bang when the recorded sequence has finished playing. Unlock your patch and

18. Create a *button* beneath *seq*
19. Connect the second outlet of *seq* to the inlet of the *button*
20. Hold the ⌘ (Mac) or ctrl (Windows) key down and click the *message start* to play back your performance[2]

Notice that the *button* connected to *seq*'s second outlet blinks when the recorded sequence has finished playing. If we use this bang to trigger the *message start*, the MIDI recording will loop. This will allow us to record a musical phrase, like an ostinato, and play something else "on top" of the ostinato as it loops continually. We should insert a *gate* between the *button* and the *message start* so that we have the option to turn off the looping feature.

21. Create a new object called *gate*
22. Connect the outlet of the *button* to the second inlet of *gate*
23. Create a *toggle*
24. Connect the outlet of *toggle* to the first inlet of *gate*
25. Connect the outlet of *gate* to the first inlet of the *message start*

2. Note that holding down the ⌘ (Mac) or ctrl (Windows) key in an unlocked patcher allows you trigger *buttons*, *message* boxes, and other objects without having to lock the patch.

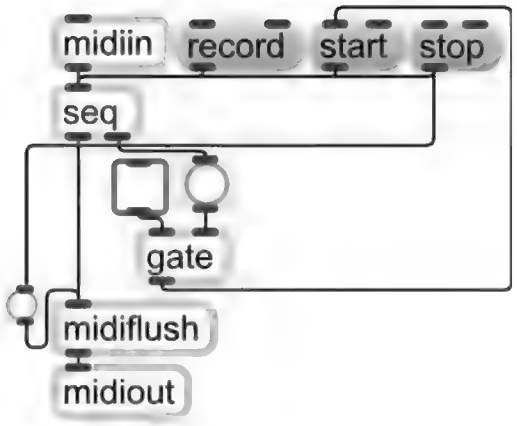


FIGURE 10.3

seq object playing in a loop only interrupted by gate | MIDI_looper .maxpat

Lock your patch and

26. Click the *message record*
27. Begin playing something on your MIDI keyboard
28. Press the *message stop* when you have finished performing
29. Click the *toggle* to turn on the looping feature
30. Press the *message start* to play back your performance in a loop
31. Play a second musical part on your MIDI keyboard as the loop continues playing
32. Click the *stop* message to stop the playback (you may also click the *toggle* again to turn off the looping feature)

Your patch now has the potential to make you a "one man (or woman) band." Let's modify this patch so that we can choose to record into one of 3 different *seq* objects with one MIDI device. This will allow us to layer parts on top of each other and experiment with musical phrases. Unlock your patch and

33. Disconnect the patch cord connecting the outlet of *midiin* to the inlet of *seq* and move *midiin* to the top of your patch
34. Highlight all other objects in the patch except for *midiin* and copy/paste them twice so that there are 3 *seq* object entities in the patch aligned horizontally
35. Create a new object called *gate* beneath *midiin* with the arguments *3* and *1*
36. Connect the outlet of *midiin* to the second inlet of *gate 3 1*
37. Create 3 *message* boxes containing the numbers *1*, 2, and *3*, respectively
38. Connect the outlet of each *message* box to the first inlet of the *gate 3 1* object
39. Create 3 new *message* boxes containing the text *record*, *start*, and *stop*, respectively
40. Connect the outlet of each *message* box to the second inlet of *gate 3 1*

41. Connect the first outlet of *gate 3 1* to the inlet of the first *seq* object
42. Connect the second outlet of *gate 3 1* to the inlet of the second *seq* object
43. Connect the third outlet of *gate 3 1* to the inlet of the third *seq* object

We need each of these *midiout* objects to operate on a different MIDI channel so that they don't interfere with each other.

44. Double click the first *midiout* object and append the argument 1 to specify MIDI channel 1
45. Double click the second *midiout* object and append the argument 2 to specify MIDI channel 2
46. Double click the third *midiout* object and append the argument 3 to specify MIDI channel 3

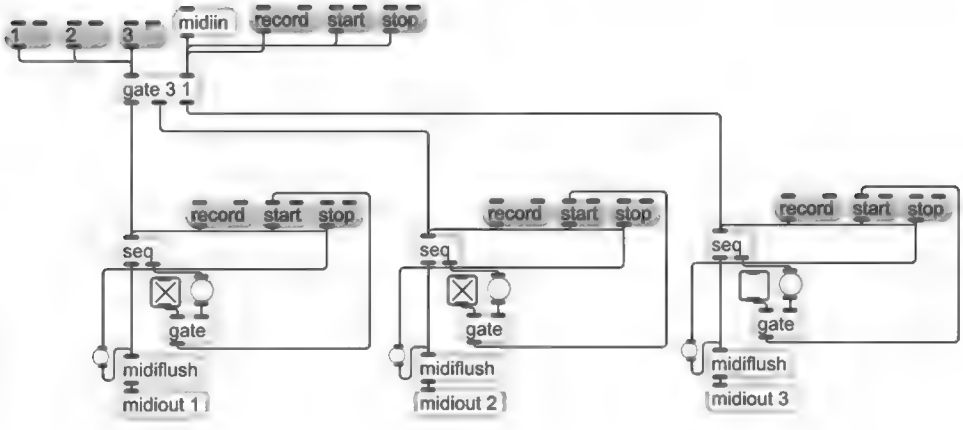


FIGURE 10.4

multiple seq entities controlled from a single midiin object | MIDI _looper.maxpat

The *gate* has 3 different locations to which it can send the MIDI info and *seq* messages. Since we gave *gate* the second argument *1*, by default, it will record into the first *seq* entity. You can now record into each different *seq* by choosing from one of the 3 numbers and clicking the appropriate *message*. Lock your patch and

47. Choose the *message 1* to send data to the first *seq* entity
48. Click the *record* message and play something on your MIDI keyboard
49. Click the *stop* and *play message* boxes (make sure that the looping *toggle* is "on" for each *seq*)
50. Choose the *message 2* to send data to the second *seq* entity
51. Click the *record* message and play something on your MIDI keyboard as the previous *seq* entity continues to loop
52. Click the *stop* and *play message* boxes (make sure that the looping *toggle* is "on" for each *seq*)
53. Choose the *message 3* to send data to the third *seq* entity

54. Click the *record* message and play something on your MIDI keyboard as the previous *seq* entity continues to loop
55. Click the *stop* and *play message* boxes (make sure that the looping *toggle* is "on" for each *seq*)
56. Click *stop* for each *seq* entity

Wow, if this activity doesn't make you want to go out and buy a MIDI foot-controller, I don't know what will! Click *File>Save* and save the file as *MIDI_looper.maxpat*. Now, let's make an advanced version of this patch with some new features. Click *File>Save As* and save the file as *MIDI_looper2.maxpat*.

A great addition to this patch would be the ability to load a single channel MIDI file (Type 0) that we can play along to. I happen to know that there is a file called *sample_ode.MID* loaded in the Max search path that was installed with the EAMIR SDK. Let's *read* this file into the first *seq* object. Unlock your patch and

57. Create a new *message* object with text *read sample_ode.MID*
58. Connect the outlet of *read sample_ode.MID* to the inlet of the first *seq* object
59. Lock your patch, make sure the *toggle* is set to loop, and click this *message* box

Notice that the MIDI file plays an arrangement of the "Ode to Joy" melody. An individual can then play along with this file, improvising a bass line or a harmonization. Let's go one step further and adjust some aspects of the tempo of this MIDI file's playback.

If we send *seq* the message *start -1*, as opposed to just *start*, we are enabling a mode by which the tempo of the sequence being played back can be altered in real time from what was initially recorded or loaded into *seq*. Unlock your patch and

60. Double click on the *message start* in the first *seq* entity and change it to read *start -1*

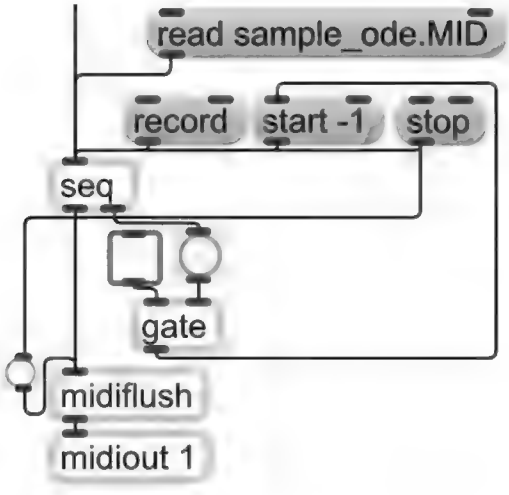


FIGURE 10.5
sample MIDI file loaded into seq | MIDI_looper2 .maxpat

To change the tempo of *seq* with *start -1*, the *seq* object must receive a series of *messages* containing the phrase *tick*. *Ticks* are the smallest increment of time understood by the computer. We can get our MIDI file to play back in a way that deals with *ticks* (as opposed to milliseconds, or bars and beats) by creating a *tempo* object and connecting its output to bang the *message tick* to *seq*.

61. Create a new *message* box containing the text *tick*
62. Connect the outlet of *message tick* to the inlet of *seq*
63. Create a new object called *tempo* with the arguments *120, 1,* and *96* (where *96* represents the smallest division of a whole note, *1*, at *120* BPMs)
64. Create a *button*
65. Connect the outlet of *tempo 120 1 96* to the inlet of the *button*
66. Connect the outlet of the *button* to the first inlet of *message tick*
67. Create a *toggle*
68. Connect the outlet of *toggle* to the first inlet of *tempo 120 1 96*
69. Create a *number* box
70. Connect the first outlet of the *number* box to the second inlet of *tempo 120 1 96*

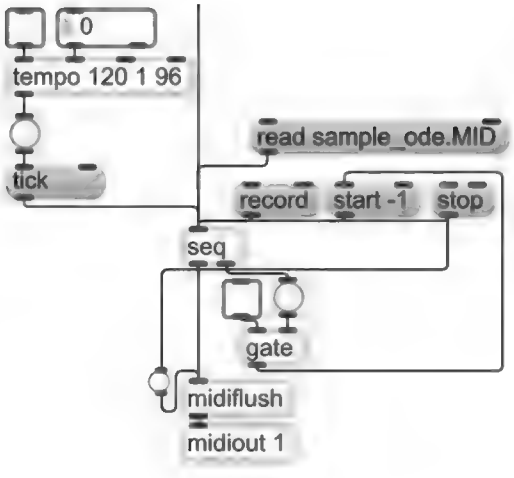


FIGURE 10.6
tempo object used to control seq playback speed | MIDI_looper2 .maxpat

71. Lock your patch and click the *message read sample_ode.MID* to ensure that it is loaded into *seq*
72. Click the *toggle* above *gate* to loop the playback
73. Click the *message start -1*
74. Click the *toggle* to turn on the *tempo* object
75. As the file plays, you may change the tempo of the playback by entering a new *Beats Per Minute* (BPM) value into the *number* box above *tempo*

Congrats! You've just added a really neat way to manipulate recorded or preexisting MIDI data. Imagine practicing along with a MIDI loop you just re-

corded, and slowing down the tempo as you improvise a second part on top of it. Can we go one step further with this patch?

Suppose you would like change the tempo of the music while it is being played back by tapping on a *button* instead of typing in a number. After all, it seems much more intuitive to tap a tempo. We can introduce a "tap tempo" into our patch that will allow us to assign a new BPM value by clicking a *button*. Sounds great, but brace yourself: there's will be a little bit of math involved.

An object called *timer* keeps track of the amount of time, in milliseconds, between receiving a bang in its first inlet and receiving a bang in its last inlet. Think of it like a stopwatch; the first *button* starts the *timer* and the second *button* reports the time between starting the *timer* and when the second *button* was pressed. As a result, if you connect a single *button* to both of *timer*'s inlets, you can report the time, in milliseconds, between each bang of the single *button*. We will use this distance between two events to create a "tap tempo" by converting the time in milliseconds to time in BPMs.

76. Create a new object called *timer*
77. Create a *button* above *timer*
78. Connect the outlet of the *button* to both inlets of *timer*
79. Create a *number* box
80. Connect the outlet of *timer* to the inlet of the *number* box

Here comes the math: this number in milliseconds needs to divide the floating point number 60,000. to give us the BPM value of one event with *timer* to the next. If we wanted to divide by 60,000, we would use the / object. However, since we want 60,000. to be divided by the milliseconds number, we will use the !/ object. This !/ object is just like the / object except that !/ receives the divisor in its first inlet and has the dividend as its argument. (Am I really using grammar school math terminology?) Don't worry: if you do these next steps correctly this one time, you can copy and paste this working bit of code for the rest of your life.

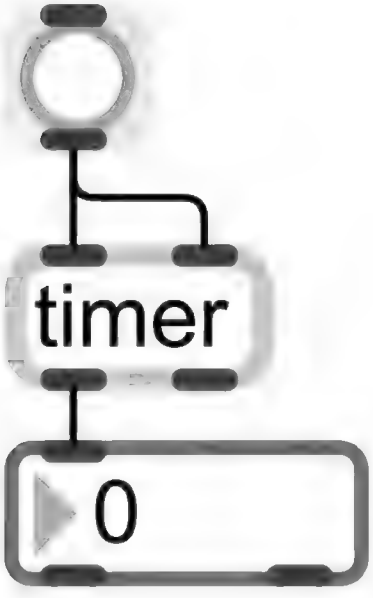


FIGURE 10.7
timer object set up to report time between two events | MIDI_looper2 .maxpat

81. Create a new object called !/ with the argument *60000.* (Don't forget the period as this is a floating point number)
82. Connect the first outlet of the *number* box receiving from *timer* to the first inlet of !/ *6000.*

83. Create a new *flonum* box
84. Connect the outlet of *!/ 6000.* to the inlet of the *flonum* box

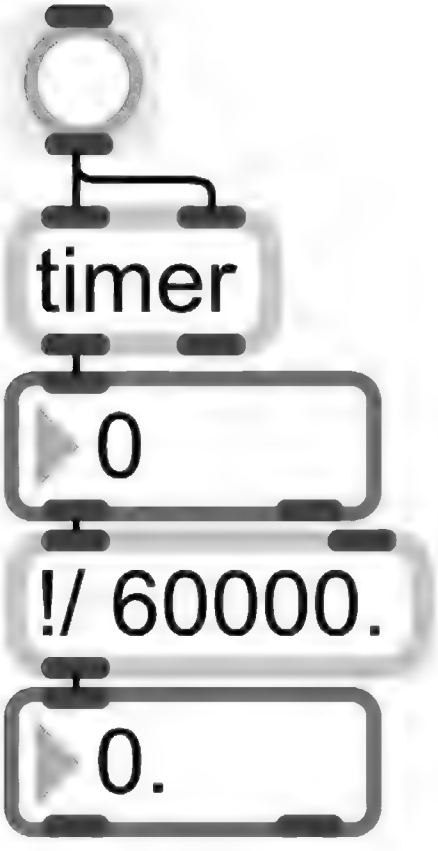


FIGURE 10.8
time in milliseconds converted into time in beats per minutes (BPM) | MIDI_looper2.maxpat

85. Lock your patch and click the *button* above *timer* repeatedly at some regular pace

The worst is over! Now, we need to make just a few adjustments to foolproof this patch. For instance, if you click the *button* once to get the tempo you want, and then get up and get a snack, the patch will continue thinking that you're adjusting the tempo. When you return to the patch and try to tap a new tempo, it will note the enormous lapse in time between the present and when the *button* was last clicked (before you got your snack) and will report a very low BPM value. To fix this, we should put some sort of rule into that patch stating that if the floating-point number is greater than or equal to some BPM, let's say *30*, then we'll accept the number as a tempo value; otherwise, don't let it pass through. Seems like a lot to ask of a single object, but the *if* object can do just that for us.

86. Create a new object called *if* with the following arguments:
$f1>=30 then $f1

You can probably already interpret how to read this. The *$f1* means a floating-point number coming through the first inlet. The whole statements reads, *if* the

floating-point number coming through inlet one (*$f1*) is greater than or equal to *30* (*>=30*) *then* send out the floating-point number coming through *$f1*. The *if* object, like *expr*, looks a lot more difficult to use than it actually is.

87. Connect the first outlet of the *flonum* receiving from !/ to the inlet of *if $f1>=30 then $f1*
88. Create a new *flonum* box
89. Connect the outlet of *if* to the inlet of the newly created *flonum* box

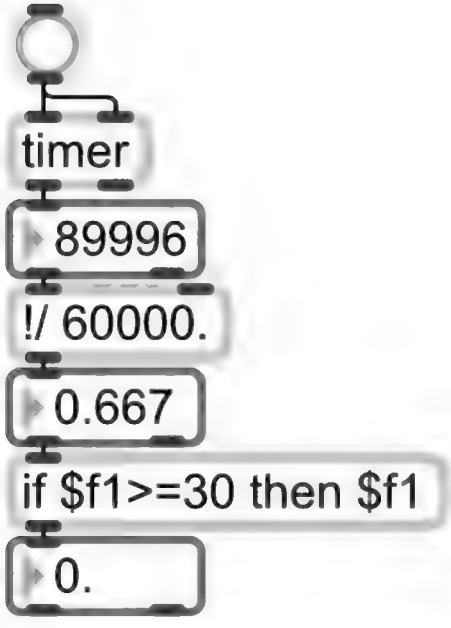


FIGURE 10.9
if object keeps tap tempo function from reporting incorrect BPM values | MIDI_looper2.maxpat

Last, let's round off that floating-point number to a real number. We can do this by adding .5 to the floating-point number and simply connecting it to a *number* box where the decimal point will be truncated. Upon having .5 added to the floating-point number, any decimal values above .5 would be bumped up the next whole number and all values lower than .5 would simply be truncated when connected to the *number* box. However, this is a great opportunity to look at the *round* object which, as you likely guessed, rounds off numbers.[3]

90. Create a new object called *round*
91. Connect the outlet of the *flonum* box receiving from *if* to the first inlet of *round*
92. Connect the outlet of *round* to the inlet of the *number* box connected to *tempo*'s second inlet (Step 70)

3. The *live.numbox* object can also be used to round numbers when it receives floating-point values.

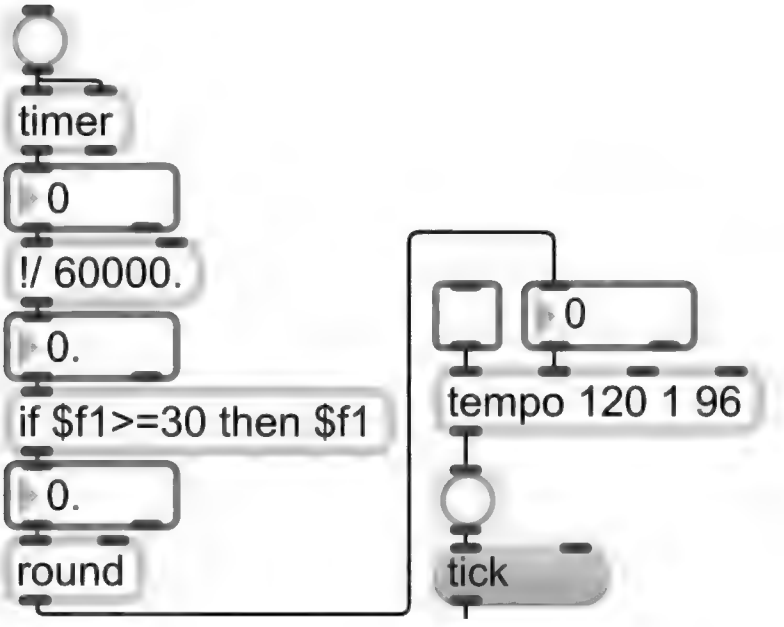


FIGURE 10.10

BPM value is rounded off and sent to tempo object | MIDI_looper2.maxpat

93. Lock your patch and click the *message read sample_ode.MID* to ensure that it is loaded into *seq*
94. Ensure that the *toggle* above *gate* is checked to loop the playback
95. Click the *message start -1*
96. Click the *toggle* to turn on the *tempo* object
97. As the file plays, you may click the *button* above *timer* to "tap" a new tempo

Imagine the possibilities of practicing conducting using a baton-like controller in which you can press a button right when the baton crosses the ictus. I bet the *Nintendo WiiRemote* would be a useful controller for this task.[4] For now, let's just assign the space bar to the "tap tempo."

98. Create a new object called *key*
99. Create a new object called *sel* with the argument *32*
100. Connect the first outlet of *key* to the first inlet of *sel 32*
101. Connect the first outlet of *sel 32* to the inlet of the *button* above *timer*

4. If you have a *Nintendo WiiRemote,* you may find the *Wii_Basics* example in the EAMIR SDK helpful.

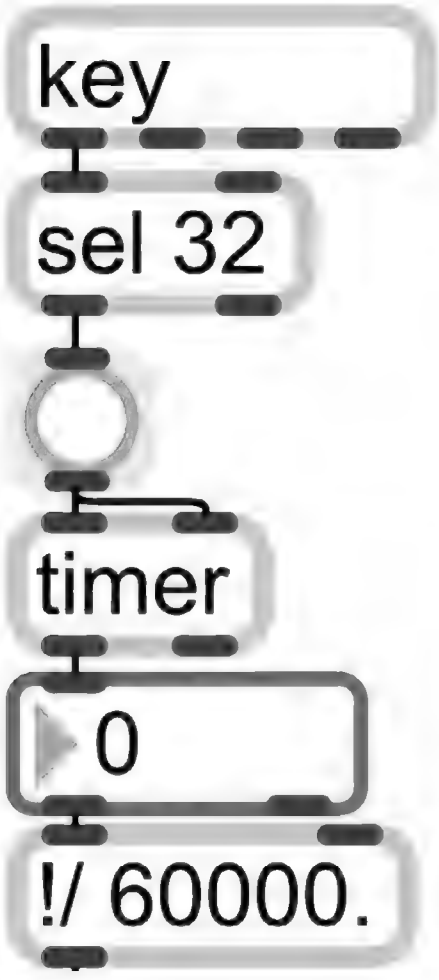


FIGURE 10.11
space bar (key 32) used to "tap" tempo | MIDI _looper2.maxpat

Click *File>Save* to save this patch then close this patch.

Step Sequencers

Step sequencers are systems that depict a measure or phrase divided into equally distanced "steps" which can have notes or other data assigned to them. The basic concept is that each step of the sequencer triggers some sound or other event to produce some sort of repeated phrase. Step sequencers are commonly used with percussion sounds but are also used with pitched instruments.

1. Open the file *select_step_sequencer.maxpat* from the *Ch 10 Examples* folder
2. Click the *toggle* to begin the sequence

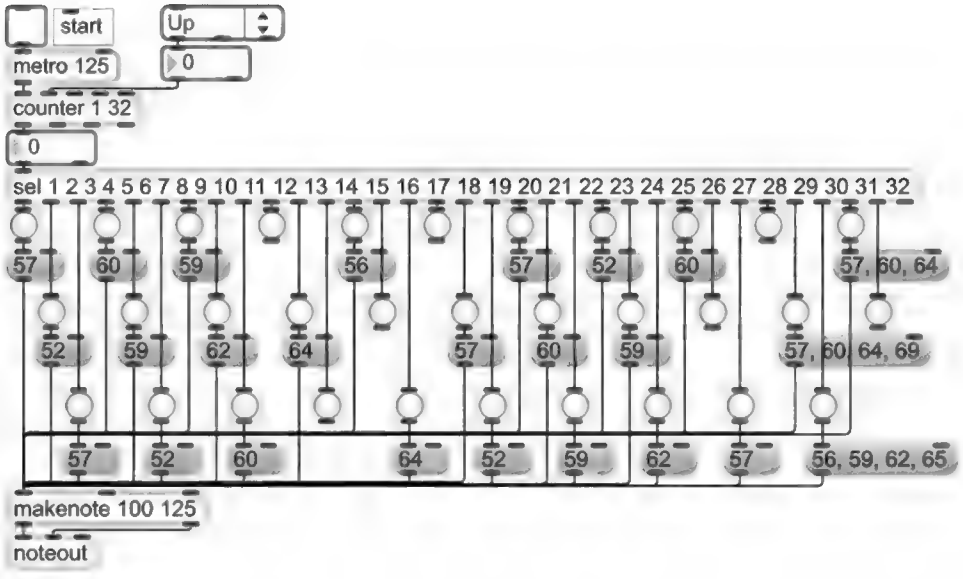


FIGURE 10.12

notes played in sequence | select_step_sequencer .maxpat

In this patch, the *metro* causes the *counter* to count from *1* to *32*. Each of the numbers from *counter* are sent to a *select* object with the arguments *1–32* causing a separate bang for each number received. Thus, there are 32 steps in this sequencer which I have mapped to some pitch numbers. Notice that some *buttons* have been intentionally left disconnected from pitches in order to add rests, which, as you know, are beats of silence.

Though you could use this *select*-based step sequencer, the Max tutorials demonstrate another helpful, and graphical, method using an object called *multislider*. Unlock the patch and

3. Highlight the *toggle, metro, counter, umenu,* and both *number* boxes and copy them
4. Create a new patch
5. Paste these copied contents into the new patch
6. Close the *select_step_sequencer* patch

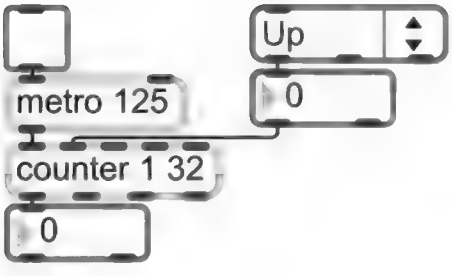


FIGURE 10.13

metro connected to counter | multislider_step _sequencer.maxpat

The *multislider* object is in many ways like the *slider* object you already know. Think of it as a bunch of *sliders* all stuck together. In this patch, we'll use *multislider* to graphically represent 3 values which we will interpret as (1) no note, (2) a kick drum, and (3) a snare drum.

7. Create a new object called *multislider*

Notice that the *multislider* even looks like a *slider*. Let's open the *Inspector* and change a few attributes to make this *multislider* better serve our patch.

8. Open the *Inspector* for *multislider*
9. In the *Inspector*, change the *Number of Sliders* to 32
10. In the *Inspector*, change the *Range* to *0.* and *2.*
11. In the *Inspector*, change the *Slider Style* to *Bar*
12. In the *Inspector*, change the *Sliders Output Values* to *Integer*
13. Close the *Inspector*
14. Click on the bottom right corner of the *multislider* and stretch it to the right to make it bigger
15. Lock your patch and click inside the *multislider* at various points

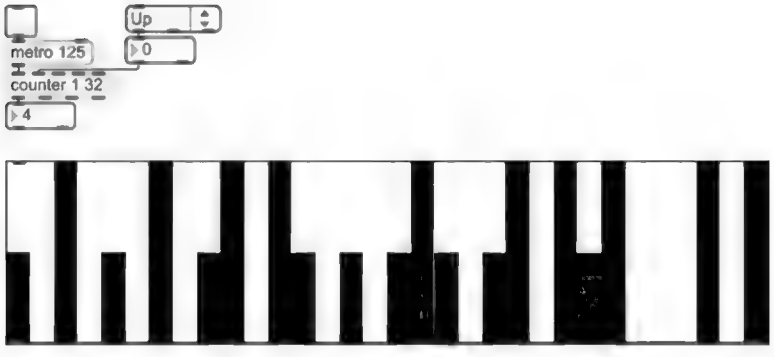


FIGURE 10.14

multislider showing 3 states for its sliders: 0, 1, or 2 | multislider_step_sequencer.maxpat

There are 3 possible states for each bar in the *multislider*: *0*, in which the bar is not visible; *1*, in which the bar is at half the height of the frame; and *2*, in which the bar is at the full height of the frame. If we had set the *multislider*'s range to something higher than *0.* to *2.* in the *Inspector*, there would be more states for each bar.

If we send *multislider* the *message fetch 1*, *multislider* will report the value (height) of bar number *1* by sending a number out of its last outlet.

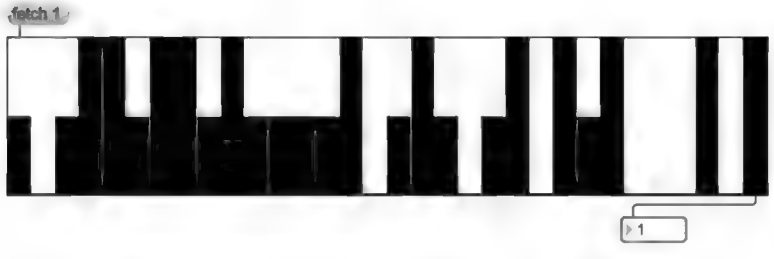


FIGURE 10.15

fetch message reporting value of slider 1 | multislider_step_sequencer.maxpat

Instead of using the *message fetch 1*, we can use the *message fetch $1* as a placeholder for the number values being generated by *counter*. This way, the *counter* object will "fetch" each value of the *multislider* as it comes from *counter*. Unlock your patch and

16. Create a new *message* box containing the text *fetch $1*
17. Connect the first outlet of the *number* box receiving from *counter* to the first inlet of the *message fetch $1*

18. Connect the outlet of the *message fetch $1* to the inlet of *multislider*
19. Create a *number* box beneath *multislider*
20. Connect the last outlet of *multislider* to the inlet of the *number* box

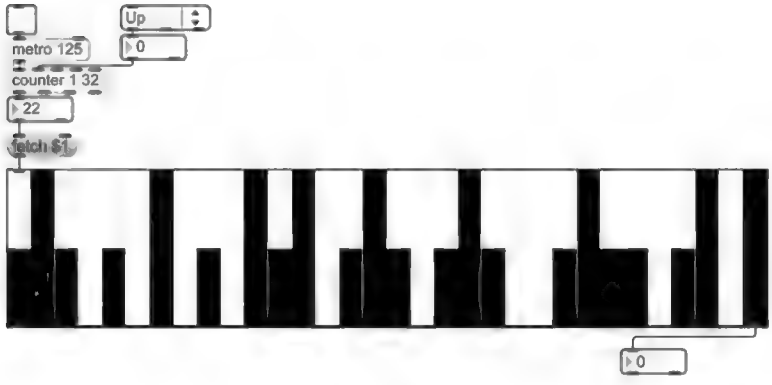


FIGURE 10.16

counter pairs with fetch to report value of each slider in multislider | multislider _step_sequencer.maxpat

21. Lock your patch and click on the *toggle* to start *metro*
22. Change the bar heights within *multislider*

Notice that the value of each bar, *0–2*, is sent to the last outlet of *multislider*. We will use a *select* object to send a bang when one of these values is received. Unlock your patch and

23. Create a new object called *select* with the arguments *1* and *2*
24. Create two *buttons*
25. Connect the first two outlets of *select 1 2* to the inlet of each *button,* respectively

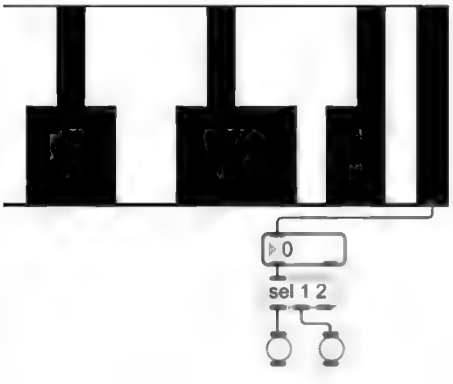


FIGURE 10.17

select detects if slider value is half height or full height | multislider_step _sequencer.maxpat

I'd like to have the values *1* and *2*, that is, bar height at half and full, to trigger a MIDI kick drum and a snare. As previously mentioned, MIDI channel 10 is reserved for percussion sounds, meaning that pitch numbers[5] played on that channel are mapped to percussion sounds when synthesized. On Channel 10, MIDI pitch *36*, which would normally be a very low C, plays a kick drum; pitch *38* plays a snare drum. Let's add some

5. General MIDI typically uses only pitches 27–87. Some other MIDI specs extend this number to include other percussive sounds.

MIDI output objects into this patch in order to use pitches *36* and *38* on MIDI channel 10.

26. Create two *message* boxes beneath both *buttons* containing the number *36* and *38,* respectively
27. Connect the outlet of each *button* to the first inlet of the *message* boxes beneath them, respectively
28. Create a new object called *makenote* with the arguments *100* and *250*
29. Connect the outlet of each *message* box to the first inlet of *makenote 100 250*
30. Create a new object called *noteout* with the argument *10* (for MIDI channel 10)
31. Connect both outlets of *makenote 100 250* to the first two inlets of *noteout 10,* respectively
32. Lock your patch and click on the *toggle* to start *metro*
33. Change the bar heights to their lowest position within *multislider* and slowly add bars of either height to the *multislider* (the lower of the two bars will be a kick drum and the highest bar will be a snare drum)

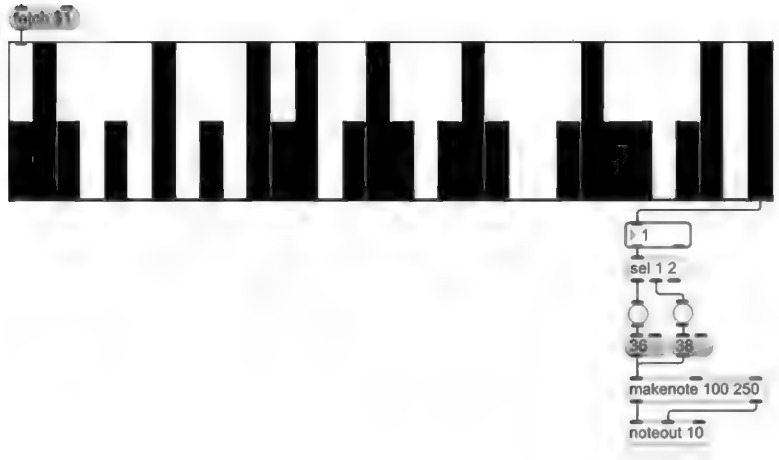


FIGURE 10.18

select is used to trigger percussion sounds | multislider_step _sequencer.maxpat

Notice that we gave *makenote* a duration that is twice the value of the *metro* speed in order to compensate for the monophonic nature of this step sequencer. Of course, we can always add more *multislider* objects to control things like hi hats and cymbals. We can even have a separate *multislider* for each of the remaining MIDI percussion sounds. For now, let's just add a simple hi hat *multislider* to the mix. Unlock your patch and

34. Copy and paste the *multislider* in your patch beneath itself
35. Connect the outlet of the *message fetch $1* to the inlet of the newly copied *multislider*

36. Highlight the *number* box beneath the original *multislider* as well as the *sel 1 2,* both *buttons*, and both *message* boxes, and copy and paste them to the right of the originals
37. Connect the last outlet of the newly copied *multislider* to the inlet of the newly copied *number* box
38. Change the values of the two newly copied *message* boxes from *36* and *38* to *44* and *46* (these are pitches for open and closed hi hat sounds on MIDI channel 10)
39. Connect the outlet of the two new *message* boxes to the first inlet of *makenote 100 250*
40. Lock your patch and click on the *toggle* to start *metro*
41. Change the bar heights within both *multislider* objects

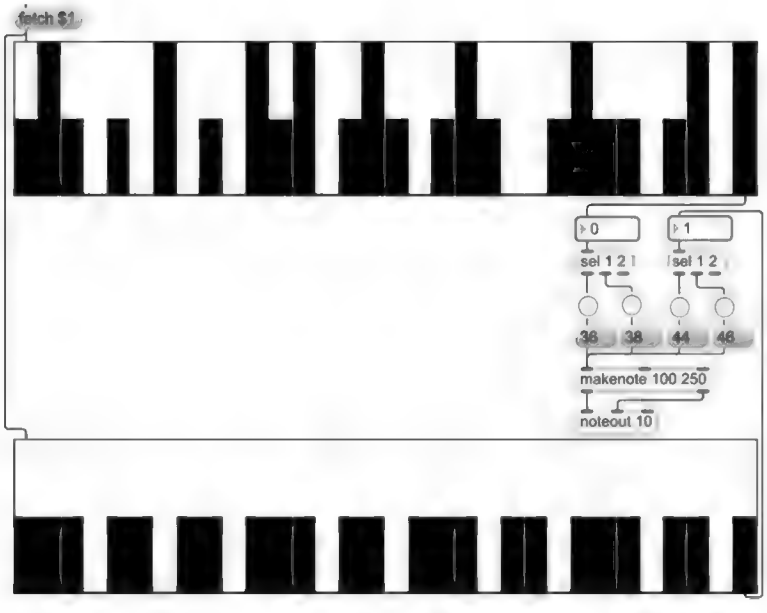


FIGURE 10.19

two multisliders controlling percussive sounds | multislider_step_sequencer.maxpat

Click *File>Save* and save this patch. Now, let's make an advanced version of this patch with some new features. Click *File>Save As* and save the file as *multislider_step_sequencer2.maxpat*.

The Transport

Max version 5 introduced a new way of dealing with time in a more global way by using the *transport* object. The *transport* object is like a master clock for handling time by which other time-based objects can be synched. In the same way that many DAWs have a transport window for adjusting tempo and time signature, the *transport* object can allow a user to make changes to the tempo and time signature and have these changes reflected in all time-based objects synched to the *transport* object. Unlock your patch and

1. If it's not still open, open the file *multislider_step_sequencer2 .maxpat*
2. Create a new object called *transport*
3. Lock your patch and double click the *transport* object to open the *transport* window



FIGURE 10.20
the transport window | multislider_step _sequencer2.maxpat

In the *transport* window, we can see the *Activate* button to turn the *transport* on as well as a *click* button to provide a clicking metronome. The controls on the left allow the user to specify *beats* per measure, the type of beat (quarter, half, whole, etc.) and the tempo (they even have a tap tempo like we created before). The controls on the right report the present location of playback in bars, beats, and units[6] since the *transport* was activated.

In the *multislider_step_sequencer2* patch, we have a *metro* object. If we want to sync this with the *transport* object, we need only to include a *note value abbreviation* in place of the millisecond value like we've been doing. Examples of *note value abbreviations* include these:

- 4n: Quarter note
- 4nd: Dotted quarter note
- 4nt: Quarter note triplet
- 8nd: Dotted eighth note
- 8n: Eighth note
- 8nt: Eighth note triplet
- 16nd: Dotted sixteenth note
- 16n: Sixteenth note
- 16nt: Sixteenth note triplet

Other note values can be used with different numbers like we used for the beat divisions of *tempo*: 1 for whole note, 2 for half note, and so on.

Our patch is a *32* step sequencer, so let's make our *metro* object beat 32nd notes synched to the *transport* object. Unlock your patch and

4. Double click the *metro* object and change its argument from *125* to *32n*

6. Units, in this and many DAW transport windows, represents "ticks" after the downbeat.

If you'll recall, we wanted the note duration of *makenote* to be twice as much as the value of *metro*, so let's change its value from *250* to *16n* (twice the duration of *32n*).

5. Double click the *makenote* object and change its argument from *250* to *16n*

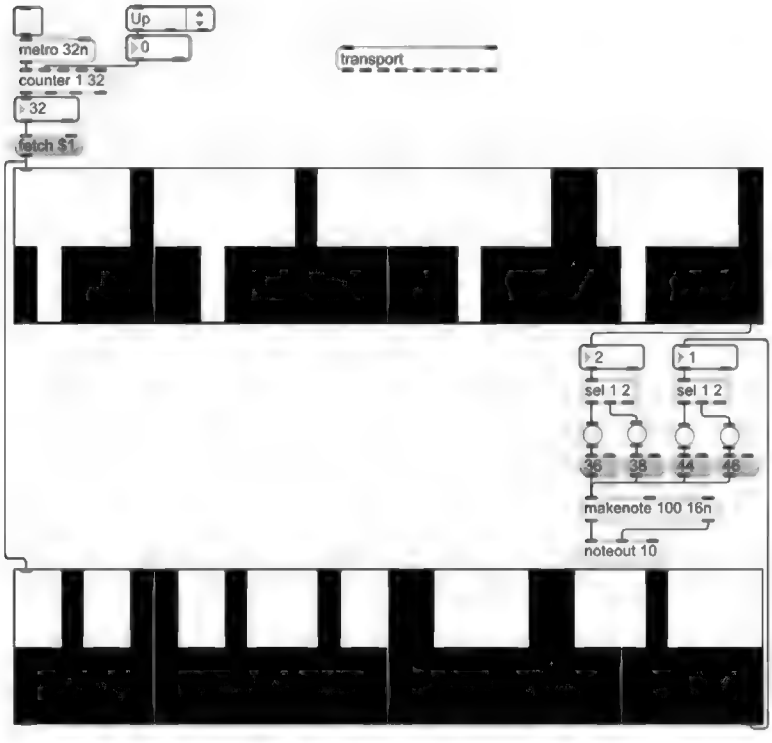


FIGURE 10.21

timing objects have been given note value abbreviations | multislider_step_sequencer2.maxpat

You can now use the *transport* window (if you accidentally closed it, just double click the *transport* object in the patch) to control the tempo of the piece.

1. Click the *Activate* button to start the *transport*
2. Adjust the tempo inside the *transport* window
3. Enable the metronome by clicking the *Click* button
4. Click the *Activate* button to turn off the *transport*

The *transport* object is incredibly useful if you are looking for a way to synchronize a bunch of time-based objects in numerous patches.

The *transport* object is also great for using Max in compositions where events occur on certain beats and measures. A related object called *timepoint* is like a *select* object dedicated to the *transport* object that takes a sequence of numbers representing bars, beats, and units, and sends a bang when that point in time is reached. Let's give *timepoint* a demonstration. Unlock your patch and

5. Create a new object called *timepoint* with the argument *4.2.0* (where *4* represents the fourth bar, *2* represents the second beat of the 4th bar, and *0* is the number of units)

6. Create a *button*
7. Connect the outlet of *timepoint 4.2.0* to the inlet of the *button*

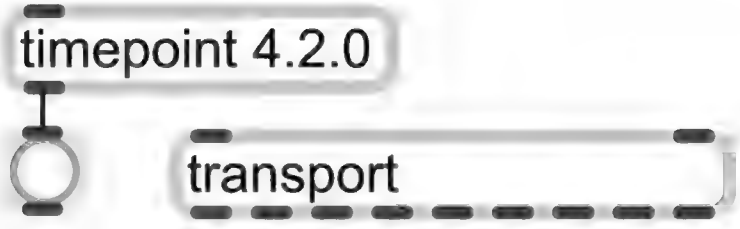


FIGURE 10.22

timepoint used to report specific times indicated by the transport | multislider_step _sequencer2.maxpat

8. Lock your patch and double click the *transport* object to open the *transport* window
9. Click the *Rewind* button in the *transport* window to set the *transport* time back to the starting values
10. Ensure that your *metro* is turned on and click the *Activate* button to start the *transport*

Notice that when the *transport* window values on the right hit the second measure of the fourth bar, the *timepoint* object recognizes that the values match its arguments and, thus, it sends a bang to the *button* beneath it.

11. Click the *Activate* button again to stop the *transport*

As mentioned, this is useful for triggering events and processes to occur at specific times. You could, for instance, trigger a set of chords in a progression to play for a certain number of bars, and use *timepoint* to trigger a different set of chords to play at a certain point. Let's use *timepoint* to trigger a beat division change to *metro*. Unlock your patch and

12. Create a new *message* box containing the text *4n*
13. Connect the outlet of the *button* receiving from *timepoint* to the first inlet of the *message 4n*
14. Connect the outlet of the *message 4n* to the second inlet of the *metro*

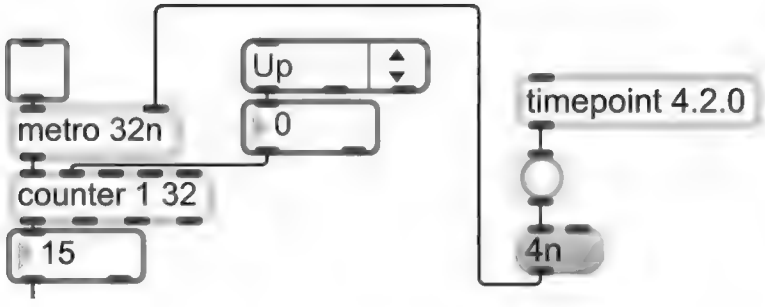


FIGURE 10.23

timepoint triggers a beat division change when transport reaches specific time | multislider_step _sequencer2.maxpat

15. Lock your patch and double click the *transport* object to open the *transport* window
16. Click the *Rewind* button in the *transport* window to set the *transport* time back to the starting values
17. Ensure that your *metro* is turned on and click the *Activate* button to start the *transport*

When the *timepoint* value is reached, a bang is sent to the *4n* message causing a beat division change for *metro*. Save and close this patch.

Overdrive

At the top menu, under *Options*, there is a setting marked *Overdrive*. Checking the *Overdrive* option enables it, which gives processing priority to timing-related operations as we've been discussing, including MIDI, over more graphical operations like mouse movements and UI objects. You can enable *Overdrive* if you are primarily using Max for MIDI and audio-related tasks. When we discuss *Jitter* and more graphic-related tasks, you should disable *Overdrive*.

Remember

- Holding down the ⌘ (Mac) or ctrl (Windows) key in an unlocked patcher allows you to trigger *buttons, message* boxes, and other objects without having to lock the patch.
- The *transport* object is like a master clock for handling time by which other objects can be synched.
- The *Overdrive* option enables it, which gives processing priority to timing-related operations as we've been discussing, including MIDI, over more graphical operations like mouse movements and UI objects.

New Objects Learned:

- *seq*
- *midiflush*
- *timer*
- *!/*
- *if*
- *round*
- *multislider*
- *transport*
- *timepoint*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Max Tutorials from the Help menu and read
  - List Processing—Manipulation of lists of data
  - Timing—Scheduling events
  - Basic Sequencing—Playing back MIDI sequence data
  - Advanced Sequencing—Recording and manipulating MIDI sequences

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that triggers "events" in some novel way. These events can be prerecorded MIDI files, chord changes, the generation of random pitches (tonal or atonal), or something else. Consider using a control interface other than typical mouse-clicking to operate the program. Consider the *transport*, *timepoint*, and other objects we've discussed in this chapter.
- Read the Help file for the objects *live.step*, *live.grid*, and *matrixctrl*



Building Stand-alone Applications

In this chapter, we will analyze a "Chord Namer" application that allows a user to enter a chord name and see the notes on a MIDI keyboard. Unlike the other patches we've worked on thus far, we will "build" this patch as a stand-alone program that can be used on any computer even if it does not have Max installed. Stand-alone programs are a great way of distributing your work to people for educational or commercial purposes.

Preparing the Application

Open the file *chord_namer.maxpat* from the *Chapter 11 Examples* folder. This patch allows users to type in the name of a chord (C, for example) and see the chord displayed on a large *kslider*. Users can then play the chord on their MIDI keyboard while looking at the visual example. The letter name of each note appears on each chord tone when it is highlighted. For taller chords, a user may enable more chord tones to be added than simply just a root, third, and fifth. For example, a user wanting to play a *Cdom7#9* chord could simply enable *7ths* and *9ths* to be displayed by checking the appropriate *toggle*s, typing *Cdom7#9* into the space provided, and pressing the *return* or *enter* key.

1. Type *C* into the text box at the top left and press the *return* or *enter* key
2. Play a C chord on your MIDI keyboard

This patch could be useful for helping people perform a piece for which they have only a lead sheet with chord names. Let's take a look inside the patch.

The patch is currently in Presentation mode. Unlock the patch and put it into Patching mode. The patch is rather large in size so you may need to zoom out on the patch (⌘ for Mac or ctrl for Windows).

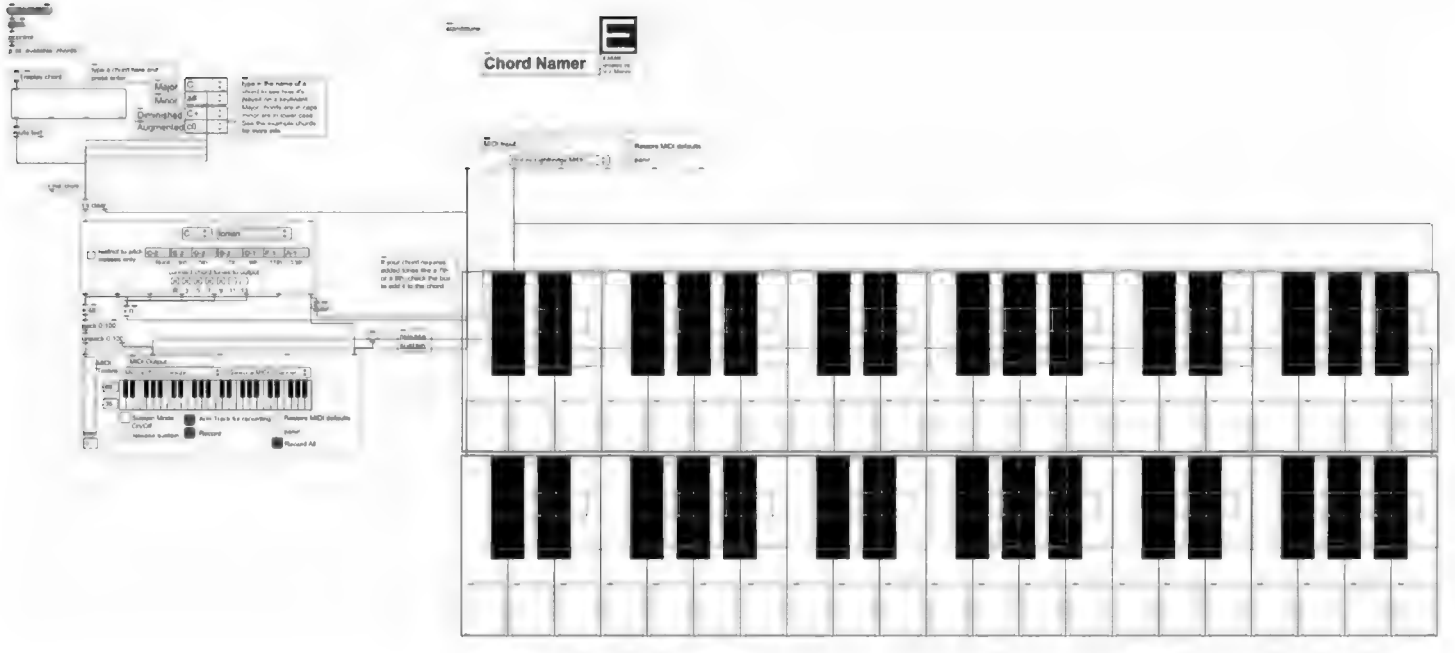


FIGURE 11.1

zoomed-out view of patch | chord_namer.maxpat

Now that the patch is open, you may be surprised to see that there is only a small number of objects inside. Take note of the 3 *bpatchers* in the patch that generate chords, handle MIDI output, and, to the right, above the *kslider*, handle MIDI input. To the far right of the patch are 2 giant *kslider* objects with *comment* boxes above each key (look familiar? Chapter 4). The top *kslider* receives pitches from the "chord generating" *bpatcher* and displays them. The lower *kslider* takes pitches from the "MIDI input" *bpatcher* and displays them.

The MIDI notes received from the "MIDI input" *bpatcher* are connected directly to the "MIDI output" *bpatcher*. The MIDI pitches received from the "chord generating" *bpatcher* have *48* added to them and are paired with the velocity value of *100* (via *pack*) before being sent to the "MIDI output" *bpatcher.*[1]

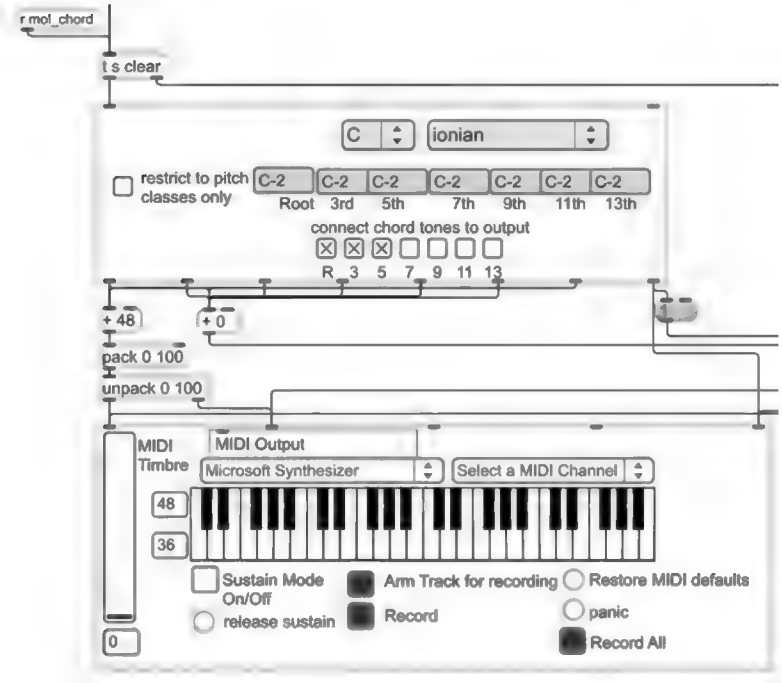


FIGURE 11.2

pitches from upper bpatcher are transposed 4 octaves and synthesized | chord_namer.maxpat

A few other points of interest in this patch are the ways by which a user may actually enter chords to be displayed. Among the methods are several *umenu* objects containing chord names connected to a *trigger (t)* object. When one of these symbols is received, *t* sends a *clear* message to the top *kslider* to remove the currently displayed chord, and then passes the new chord message to the chord-generating *bpatcher*. This *bpatcher*, in turn, displays the new chord and sends the appropriate MIDI notes to the MIDI output *bpatcher*.

1. Notice that the generated chords have *0* added so that they are displayed as pitch classes in the *kslider*. A velocity *message* of *1* is also sent to the *kslider* in order to display the pitches (since *kslider* needs both a pitch and velocity value). This, of course, has no bearing on the notes we hear because the top *kslider* is not connected to the "MIDI output" *bpatcher*.

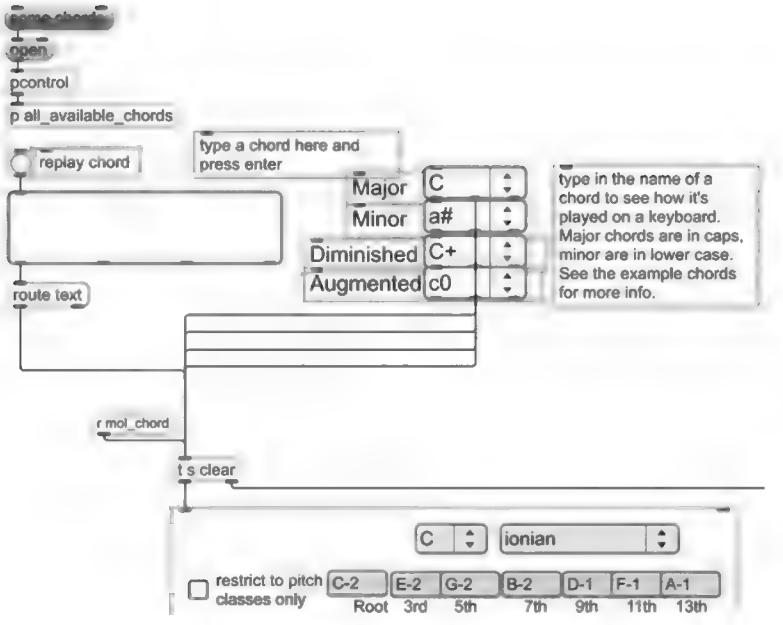


FIGURE 11.3

chords can be entered manually or selected from umenu objects | chord _namer.maxpat

A *message* box labeled *some chords* bangs an *open* message to an object called *pcontrol*. The *pcontrol* object communicates with an object called *thispatcher* inside of the subpatcher named *all_available_chords*.

3. Hold ⌘ (Mac) or ctrl (Windows) and click the *some chords* message to open the subpatcher named *all_available_chords*
4. Unlock the window that opens and choose *View>Presentation* from the top menu to exit Presentation mode (Note: it is necessary to unlock the patch and exit Presentation mode via the *View* menu at the top or with key commands since this subpatcher does not display the bottom menu)

In Patching mode, the subpatcher window reveals an *inlet* connected directly to a *thispatcher* object. The *thispatcher* object is useful for sending messages to a patcher to resize the patcher window, remove the scrollbars from a patcher, open the patcher in full screen and perform more actions on the patcher itself. *Thispatcher* is a very useful object for customizing a patcher window in real time. In this case, since the *thispatcher* is in a subpatcher, we need to use the object *pcontrol* to communicate with the *thispatcher*. For this reason, we've sent it the message *open* which will be received by *thispatcher* and will open the patcher window.

5. Close the subpatcher window

The window that opens simply reveals a bunch of other *umenu* objects displaying chord names and functions. Each middle outlet of *umenu* is connected to a *send* object (*s*) named *mol_chord*. The chord messages are received by a *receive*

(*r*) with the same name which is connected to the "chord generation" *bpatcher* via the *t* object.

The most obvious way of getting chords into the "chord generation" *bpatcher* is through the large *textedit* object in the middle of the screen. This object allows the user to enter text, such as chord names, into a box. When the user presses the enter or return key, the text in the box is sent out of *textedit*'s first outlet prepended with the word *text*. For example, *Cdom7* chord entered in *textedit* would be outputted as "*text Cdom7*." It is for this reason that the *route* object with the argument *text* is being used.

The text from *textedit* will be sent to *route* where the prepended word *text* will be removed, allowing the remainder of the message to be sent to chord generation *bpatcher* via the *t* object. The *button* connected to the *textedit* object simply resends the same text inside of *textedit* repeating the process involving *route*.

The more useful features in this patch have been added to the Presentation view and the patch has been set to open in Presentation mode in the *Patcher Inspector*.

6. Put the patch in Presentation mode
7. Make sure that the patch is viewed at 100% and not zoomed in or out (⌘+*1* for Mac or ctrl+*1* for Windows)
8. Resize the patcher window to the desired size that shows the controls and the *kslider* but not a lot of extra white space

Let's open the Patcher Inspector and Change a few more things:

9. Open the Patcher Inspector and change the *Background Color* to a color of your choosing. You will want to choose a color that will allow the text to be readable. You may also want to avoid colors that might be tiresome to look at for a long period of time
10. Uncheck the boxes named *Show Horizontal Scrollbar* and *Show Vertical Scrollbar* since our patch is already resized to the dimensions we like (Note: if your computer monitor does not show the entire patch in the screen in Presentation mode without scrolling, skip this step)
11. Close the Inspector
12. Click *File>Save* to save the patch as it is

To customize the default colors used in your patch, you can select *Options>Object Defaults* from the top menu. From here you can manually set the default values for the different patch components. You can also load one of the default color templates by selecting the icon in the lower left corner of the *Object Defaults* window.

Building the Application

The patch is ready to be built as a stand-alone application. To do so, choose *File>Build Collective / Application* from the top menu.



FIGURE 11.4
the Build window

In the *Build* window, you will notice that the text *open thispatcher* has been added to the "Script." This script is the list of things that will get merged into a single file when we turn this patch into a stand-alone application. Clicking the "Build" button makes a single file of your patch and the Max *Runtime* files, files that allow Max patches to be run, but not edited. As long as the free Max *Runtime* files[2] are installed on their computers, you can currently give your patches to others, who can open and run them even if they don't have Max installed. Despite this, it's sometimes a lot easier to give someone a stand-alone application that has your Max patch merged together with the *Runtime* files.

Before we can click the "Build" button, we need to make sure that all of the objects, images, abstractions, and other files associated with this patch have been added to the script. For example, we are using objects and patches in the EAMIR SDK (all of those *bpatcher*s, for example), so we must add this folder to the script.

13. Click *Include Folder* and browse to find the folder called *EAMIR_SDK* on your computer[3]
    a. On a Mac, this folder was installed to HD:\Applications\Max 5\Cycling '74 when you ran the installer
    b. On Windows, this folder was installed to C:\Program Files\Cycling '74\Max 5.0\Cycling '74 on 32-bit Windows or C:\Program Files (x86)\Cycling '74\Max 5.0\Cycling '74 on 64-bit Windows
14. When you find the folder, highlight it and click OK. You will see the folder added to the script

In Max 6, the "Save As Project" option from the File menu will automatically bundle the patcher and other dependent files inside of a project folder. In both Max 5 and 6, there is no obligation to add any of the objects that come with Max to the script as they will automatically be included in the build. However, if you

2. The Max Runtime files are available for free from http://www.cycling74.com.

3. The Max application directory may differ depending on your version of Max. If you manually installed the EAMIR SDK, you will have to manually locate the folder on your computer.

ever notice that for some reason a Max object does not load correctly, or you get an error in the Max window after you build the application, you may add these objects in the same way that we added the EAMIR SDK. You can determine what external objects and patchers are being used in your patch by clicking *File>List Externals and Subpatcher Files* from the top menu. The files in use will be displayed in the Max window. If multiple items are displayed, but located in the same folder, use *Include Folder* to copy the folder to the build.

It looks like everything is in order, so go ahead and

15. Click the "Build" button
16. Choose a location where you want to save the application
17. Type in a name for this file

You have the option to build this program as a *Collective* or as an *Application*. A *collective* can only be run with the Max *Runtime* file; however, unlike a patch, it cannot be unlocked. *Collectives* are smaller in file size than *applications*. For our purposes, we will build this program as an *application* which will have the *Runtime* file embedded in it.

18. Select *Application* from the pull-down menu
19. Click *Save* to start the build process. The *Build* screen will save "Finished Script" when the build process is complete

Congrats! Your application has been built.

20. Click *Save Script* from the *Build* screen and save the script as *chord_name_script.txt*. This allows us to reuse the same script later on if we have to rebuild the patch.[4]
21. Close the *Build* screen
22. Locate the file at the location on your computer where you saved it and double click it to open it.

When you open the application, one thing you will notice is that the Max window is visible. While this isn't the worst thing in the world, we'll discuss a way to hide it in a moment.[5] You may also may want to change the icon of your application. We'll rebuild our patch and address those issues momentarily. Please know that it is perfectly normal to build an application, realize that something doesn't look or work right, and then rebuild it again after you've worked the "bugs" out. This is all part of the "debugging" process. For this reason, it's best practice to test out your applications thoroughly after you build them. When you are certain that everything is working correctly, then you can share your work with others.

4. Since it's perfectly normal to build an application and then realize that you need to change something, it is a time-saver to save the script for your app so that you don't have to repeat the same "Include File/Folder" steps each time you fix a typo.

5. If there are errors in your Max window highlighted in red, repeat the above steps and rebuild your application.

Before we go trashing this application, take a quick look at *umenu* in the MIDI Output section. You may have noticed the option "from MaxMSP 1" and "from MaxMSP 2."[6] These are two virtual MIDI ports provided by Max for routing your MIDI data to other applications. Since MIDI is essentially a numerical way of representing musical elements, you can route these numbers to different synthesizers if you'd prefer not to use the internal synthesizer sounds provided for you by your computer. For instance, suppose you just love the electric guitar sound in Garageband or Kontakt and you want the notes in your patch to be synthesized with those sounds. Selecting "from MaxMSP 1" will send MIDI data out of this virtual MIDI port instead of synthesizing it internally as we've been doing so far. You can then open any stand-alone synth program or DAW and, set the MIDI input preferences for that program to receive MIDI from "MaxMSP1" as if your program was an actual MIDI hardware device connected to your computer. You can then generate MIDI data in Max and have it synthesized with the timbres of your favorite synth programs. Remember, the MIDI data are just numbers and you can route them to any timbre you please.

23. Close this application
24. Get back into your Max patch
25. Unlock the patch and put it in Patching mode
26. Create a new object called *standalone*
27. Open the *Inspector* for *standalone*

The *standalone* object is used for specifying some particulars about the stand-alone application you are building. By default, some of the settings are checked such as *Audio Support* and *Can't Close Top Level Patcher*. There are other options in the *Inspector* that are worth mentioning such as the option to have *Overdrive* enabled by default. Since I mentioned my reluctance to have the Max window visible when the application loaded, we will uncheck the option here.

28. Uncheck the option *Status Window Visible at Startup*
29. Check the option *Enable Overdrive*
30. Close the *Inspector*
31. Choose *File>Build Collective / Application* from the top menu
32. Click *Open Script* and locate the script you saved during the previous build attempt (if you forgot to do this step, see Steps 13–14) and click *Open*

Let's add a custom icon to our application. To do so, we'll simply add the icon file as we would any other file. There are two EAMIR icons in the *Chapter 11 Examples* folder: *EAMIR.ico* for Windows users and *EAMIR_icon.icns* for Mac users.

6. This feature may not be available for Windows, but third party virtual MIDI port software like *LoopBe1* from http://nerds.de/ and *MIDI Yoke* from http://www.midiox.com/ are available and work similarly.

33. Click *Include File* from the *Build* screen
34. Locate the appropriate icon for your platform from the *Chapter 11 Examples* folder and click *Open*
35. Click in the *Script* box and change the word *include* to *appicon* in front of the path to the icon you just located by highlighting and replacing the text
36. Click the "Build" button
37. Choose a location where you want to save the application
38. Type in a name for this file
39. Select *Application* from the pull-down menu
40. Click *Save* to start the build process
41. Click *Save Script* from *Build* screen and save the script as *chord_name_script2.txt*
42. Close the *Build* screen
43. Locate the file at the location on your computer where you saved it and double click it to open it (Note its snazzy new icon)

The application should load without problems.

Icons

There are several ways to get icons for your applications. One is to search on the Internet for free icons. The web is full of Windows formatted *.ico* files. Another way is to design them yourself with icon creation software. There are numerous free Windows software products to allow custom icon creation.

On a Mac, the free development tool *Xcode* (included on your Mac OS disc) has a Utility called *Icon Composer* in the *Developer/Applications/Utilities* folder that can be used to create *.icns* files. However, you can also change the icon for any Mac application or file by opening an image in *Preview*, highlighting and copying all or part of the image file you want to use as your icon, choosing *Get Info* on the file whose icon you want to change, clicking on the icon image at the top left of the *Get Info* screen, and pasting the copied image over the existing icon.

Just as I mentioned when we discussed loading images into your patch through *fpic*, an icon with your favorite celebrity, cartoon, or sports hero may be the difference between a boring activity and a cool one.

Permission and Cross-platform Building

Remember that you should always get permission to use something that isn't yours: images, sound files, MIDI files, and so on. For example, you are certainly able to make a stand-alone application of Max objects and sell it for as much as someone will pay for it. However, if you use third party externals and

abstractions or a patch that you found on the Cycling '74 forum, you must get permission.

Some objects and libraries have license agreements explaining the terms of use for the software. The EAMIR SDK and the Modal Object Library, for instance, use the Creative Commons (CC) license agreement[7] which permits noncommercial use under certain terms. Some authors will allow you to pay a licensing fee for the right to include their objects in a commercial product. Contacting the author is the best way to find out what is and isn't allowed when it comes to using work that isn't yours.

A Max patch that you create on a Mac will open and run without problems on Windows with the exception of a few built-in objects that are platform-proprietary (we haven't discussed any of these yet) and any external objects that you use likely have Mac and Windows versions (the Modal Object Library, for instance, has Mac and Windows versions of each object in the same folder). However, the stand-alone application you build on a Mac will not run on Windows and vice versa. If you build a stand-alone application on a Mac and want to build it to run on Windows, you will need to find a Windows machine running Max and repeat the build process on that machine. Consult the Max window during this process to ensure that you don't get any errors. Close the Max patch.

Remember

- To customize the default colors used in your patch, you can select *Options>Object Defaults* from the top menu.
- It's best practice to test out your applications after you build them.
- You can route MIDI data "from MaxMSP 1" and receive it in a program like Garageband to use its sound library instead of the General MIDI timbres.

New Objects Learned:

- *pcontrol*
- *thispatcher*
- *textedit*
- *standalone*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

7. Attribution-NonCommercial-ShareAlike 3.0 Unported http://creativecommons.org/licenses/by-nc-sa/3.0/.

- Select Max Help from the Help menu and read
  - Sharing Patches with Others
- Select Max Tutorials from the Help menu and read:
  - Presentation Mode—Creating a presentation interface for a patcher

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Make stand-alone applications for one of the previous assignments you completed or open the file *building_apps.maxpat* from within the *Chapter 11 Examples* folder and build a new standalone based on this patch. Be sure to include any external objects or files used in the patch while making the build.
- Examine the KeyViewer application in the *Chapter 11 Examples* folder.

12

Working with Audio

In this chapter, we will discuss MSP, a collection of objects that work with audio signals. Unlike the Max objects we've discussed so far that handled data like numbers (including MIDI) and text, the MSP objects can handle actual sound recordings, like audio from a microphone, as well as generate signals. As we will see, MSP objects have a ~ after their name and slightly different colored patch cords signifying that they are handling some sort of audio signal.

By the end of this chapter, you will be able to get a microphone signal in and out of Max, generate timbre according to the harmonic series, and build your very own synthesizer from that timbre.

Basic Ins and Outs of Audio

Let's begin by building a basic patch that takes input from a microphone and sends it out to your speakers. Your computer is a digital device; you, as I hope you know, are not. Your voice is an acoustic instrument which, in the early days of recording, used to be recorded using analog recording techniques to represent qualities of the acoustic instrument. In order for your voice to be recorded and represented by a computer, there needs to be some conversion of the analog signal to a digital signal.

For this reason, your computer's sound card has an analog to digital converter, otherwise known as an A to D converter or, simply an ADC. Depending

on the type of sound card you have, some AD converters will do a better job than others of supplying enough digits to represent the analog sound being recorded. Think about it like this: if you were allowed 50 words to describe your favorite food, the description would be a lot more articulate than if you had only 15 words to describe your favorite food. In the same regard, in a digital recording system, the more digits you have to represent the analog signal you are recording, the closer the representation will be to the original source you can hear with your ears.

For now, we will leave this discussion of analog to digital conversion. Just know that when we record something, we are making a numerical representation of what we hear by using a microphone connected to a computer similar to the way we make a mental representation of what we hear by using our ears that are connected to our brain. The MSP object for obtaining audio from your computer is the *adc~* object.

1. Create a new patch
2. Create a new object called *adc~*
3. Create a *toggle* above *adc~*
4. Connect the outlet of *toggle* to the inlet of *adc~*

When the *toggle* is turned on, the *adc~* object will begin taking in audio from your computer's sound card. The audio signal received can then be sent to other MSP objects, though currently we won't hear anything because *adc~* is not connected to any other objects. By default, *adc~* is receiving audio on audio input channels *1* and *2* (left and right) of your computer's stereo sound card. If your sound card has numerous microphone inputs, you can specify any number of arguments for *adc~* to obtain audio on those channels instead. In fact, if you have a 4 channel soundcard, you can give *adc~* the arguments *1 2 3 4* to obtain a signal on all 4 channels. The outlets of *adc~* object would then change from *2* to *4*. For information on the sound card connected to your computer and how those inputs and outputs are organized in Max, choose *Options>DSP Status* from the top menu. You can also get to the DSP Status by double clicking the *adc~* object or by sending *adc~* the *message open*.[1]

Note that the *adc~* object has an inlet that works with regular non-MSP Max objects even though *adc~*'s output is going to be a signal that will not be understood by regular non-MSP Max objects. This type of interchange is common for MSP objects. Let's create an object called *meter~* that will let us know if we are actually getting audio signal in from our microphone.

5. Create a new object called *meter~* beneath *adc~*
6. Connect both outlets of *adc~* to the first inlet of *meter~*

1. If you have trouble getting audio to play back in Max or simply want to test your audio, you may choose *Audiotester* from the *Extras* menu at the top menu.

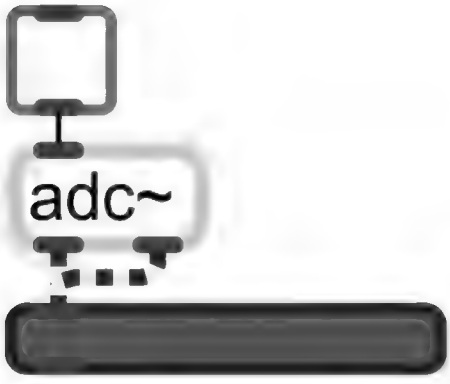


FIGURE 12.1
audio input connected to a meter

Notice that the patch cord connecting *adc~* to *meter~* looks different from the patch cord connecting the *toggle* to the *adc~*. This patch cord is signifying that there is some sort of audio signal being sent through it as opposed to numbers and text.

7. Lock your patch and click on the *toggle*
8. Begin speaking into your computer's microphone to see the *meter~* light up (if speaking into your microphone does not seem to be functioning with *meter~*, check the Max window for any errors, and then choose *Options>DSP Status* from the top menu to ensure that the correct sound card has been chosen from the *Driver* menu and that the audio channel your microphone is connected to is set as either *Channel 1* or *Channel 2*)

As mentioned, the two outlets on *adc~* are input channels *1* and *2*, that is, left and right if you're using a stereo microphone. We have both outputs going to a single *meter~* object for the purpose of ensuring that our microphone is working. Let's now take steps to get this input signal back out to our speakers. Before we do that, we should discuss some safety precautions that you should take.

If you buy a commercial hardware recording system, or some sort of hardware effects unit, there is little chance that you could ever do something to the device in error that would cause it to start generating distorted noise at deafening levels. It's not to say that there aren't user errors that cause moments of loud noise, but there are certain precautions that are built into those systems to ensure that they work a certain way.

In Max, however, *you* are the manufacturer of the system. If your system has precautions, it would have to be because you put them in there. Because Max provides so much freedom to its users, if your system blows up your computer speakers, Max has no idea that this is a bad thing and will assume that you are knowingly doing this (performance art?). With that said, make sure that you have easy access to the volume knob on your speakers or sound card while you are working with audio signals, particularly if you are working with headphones. It's easy to forget that you left something on and accidentally generate sound.

In order to add a level of protection, we will use the *gain~* object to ensure that we control how much of the signal we're going to work with. Unlock your patch and

9. Create 2 new objects called *gain~* beneath *adc~*
10. Connect each outlet of *adc~* to the first inlet of each *gain~* object. respectively (be sure that you are connecting the patch cords to the first inlet of each *gain~* object)

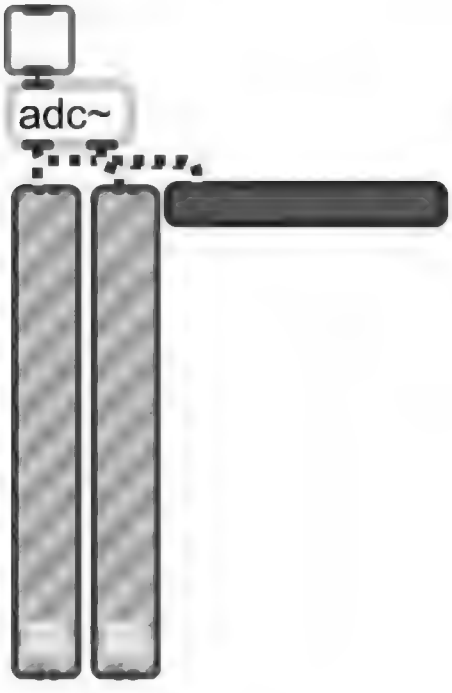


FIGURE 12.2
stereo audio input with gain control | audio_ins .maxpat

The *gain~* object looks similar to the *slider* object. It is a graphical object for reducing or increasing the amount of signal flow from the inlet source (first inlet) to the outlet destination (first outlet). To represent the increments of the *gain~* object, let's connect some *number* boxes to its second outlet. This might also be a good time to resize the *meter~* object so that it stands vertically alongside these *gain~* objects.

11. Create 2 *number* boxes beneath each *gain~*
12. Connect the second outlet of each *gain~* to the inlet of the *number* box beneath it
13. Click the bottom right of the *meter~* object and resize it so that it stands vertically alongside the *gain~* object

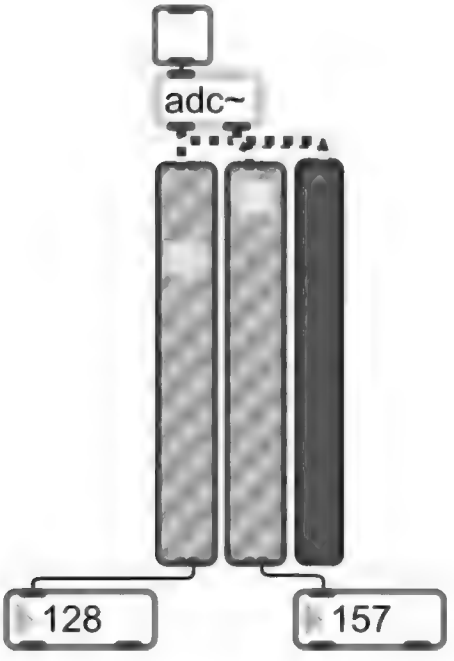


FIGURE 12.3

gain~ objects displaying their level value numerically | audio_ins.maxpat

14. Lock your patch and move the sliders on the *gain~* object

When the slider value of *gain~* is at *128*, it's as if the signal is being multiplied by *1*: there is no increase or decrease in the gain level. This level is referred to as "unity gain." Of course, there will likely be times when you want to add gain to your signals, but for our purposes, don't let the *gain~* object exceed *128*. One way we can ensure that the *gain~* will not exceed *128* is by adding a single *slider* object to control both *gain~* objects. Unlock your patch and

15. Create a new *slider* object
16. Connect the outlet of *slider* to the first inlet of both *gain~* objects

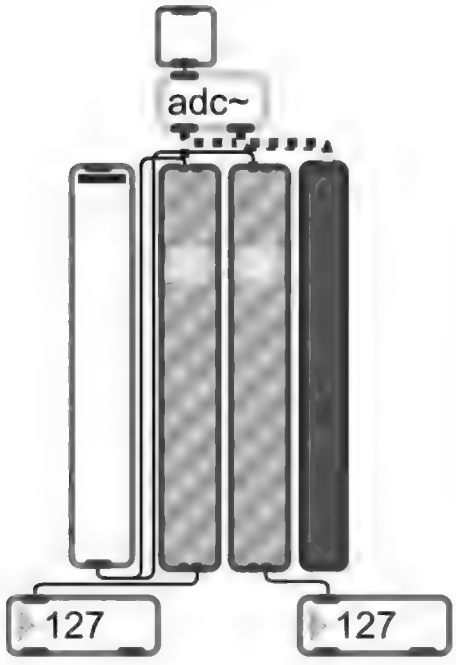


FIGURE 12.4

a single slider object controls both gain~ levels | audio_ins.maxpat

Since the default maximum output of *slider* is *127*, our *gain~* objects will not exceed *127* as long as they're being controlled with the *slider*.

Now, let's add the final piece of this patch that will allow us to hear what is coming through our mic: the *dac~* object. "DAC" stands for Digital to Analog Converter. The *dac~* object is the reverse process of *adc~*. Again, if your computer has numerous output channels, you can specify them as arguments, to *dac~*. By default, *dac~* will operate on output channels *1* and *2* and, thus, will have two inlets which will be interpreted by stereo sound cards as *left* and *right* audio channels.

17. Create a new object called *dac~*
18. Connect the first outlet of each *gain~* to the 2 inlets of *dac~*, respectively (Note: be careful to connect the first outlet of *gain~* to the *dac~*. Remember that the color of the patch cord carrying an audio signal will be different from solid black patch cords carrying numbers and text)
19. Create a *toggle* above *dac~*
20. Connect the outlet of the *toggle* to the first inlet of *dac~*

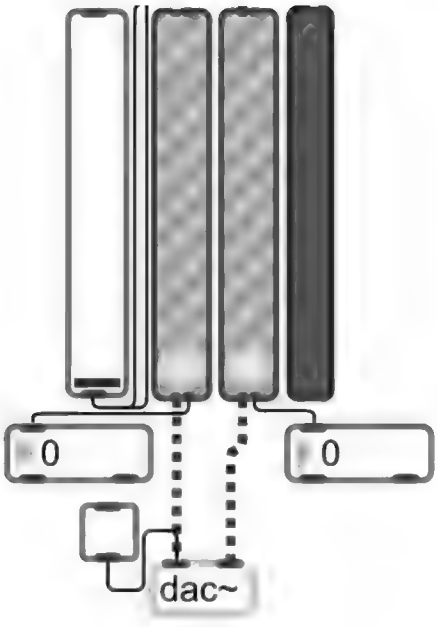


FIGURE 12.5
audio input entity connects to audio output object | audio_ins.maxpat

21. Lock your patch, lower your *slider* value to *0*, turn on both *toggle* objects, and speak into your microphone while slowly increasing the *slider* value (Note: be careful that your microphone is not positioned near your speakers or you will get feedback)

These are the basics of getting audio in and out of Max. In the future, since we are only going to be working with stereo input, we will use the objects *ezadc~* and *ezdac~* as they are a more graphical and intuitive version of *adc~* and *dac~*. Click *File>Save* and save this patch as *audio_ins.maxpat*. Close this patch.

Sine Waves

A sinusoid, as you may remember from your fifth grade science class, is an oscillating wave. It is sometimes referred to as a sine wave, and in music, is referenced often in terms of cycles per second, or hertz (Hz). A sine wave that oscillates 440 times per second is said to move at *440 Hz.*

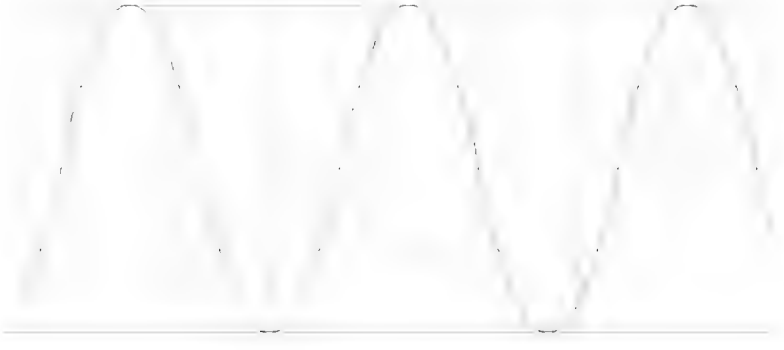


FIGURE 12.6
snapshot of oscillating sine wave

The range of hearing for humans is somewhere around 20Hz–20,000 Hz, or 20 kilohertz (kHz) although it's difficult to find people who can hear the extremes of this range. When a sine wave oscillates with a fast enough frequency, we can hear the frequency as having a definite pitch. For example, *440 Hz* is often used as the reference frequency for the *A* above middle C. In fact, if you swung your coworker over your head *440* times in one second, amid the cries of terror, you would hear your coworker's body generate the pitch we refer to as *A*. The letter names we attribute to these frequencies are abstract. As you know, some orchestras tune to *438 Hz* and call it an *A*. One reason that the letter names for pitches are used is because it's a lot easier to tell someone to play *A*, *B♭*, and *C♯* than *440 Hz, 466.2 Hz,* and *554.4 Hz.*

Some pitches at the low end of the hearing range are felt as a rumble more than they are heard as a pitch. I used to tell my middle school students to match the pitches I played by singing them. I would then play a few notes on the piano and a few on the guitar in different octaves for which the students would attempt to sing back the pitches. As you would expect, when I started playing the kick drum, the students had trouble singling out those pitches and would start imitating "boomy" noises and what sounded at times like explosions. The difficulty in their task was not only that the frequency was so low that they couldn't physically sing it, but also that it was too low for them to determine the sound as having a definite pitch. The point of this illustration is that some sounds move at such a slow speed that they are not heard as having definite pitch that one could imitate by singing. We refer to these sounds as having an indefinite pitch.

All right, let's make some sine waves in Max. To generate sine waves, we will use the object *cycle~* to convert floating-point numbers into cycles per second. We will use floating-point numbers instead of integers because the floating-point numbers allow for greater precision.

1. Create a new patch
2. Create a new object called *cycle~*
3. Create a new *flonum* box
4. Connect the first outlet of the *flonum* box to the first inlet of *cycle~*
5. Create a new object called *gain~*
6. Connect the outlet of *cycle~* to the first inlet of *gain~*
7. Create a new object called *ezdac~*
8. Connect the first outlet of *gain~* to both inlets of the *ezdac~* object



FIGURE 12.7

cycle~ connected to ezdac~ with gain control | sine_stuff.maxpat

9. Lock your patch and turn on the *ezdac~* by clicking on it (you will see the icon turn from a light gray color to a shade of light blue)
10. Type the number *440* in the *flonum* box
11. Slowly increase the *gain~* value until you hear the frequency *440 Hz*
12. Change the numbers inside the *flonum* box to hear different frequencies
13. Turn off the *ezdac~* by clicking the icon again

Let's use an object called *scope~* to graphically represent the sine wave. Unlock your patch and

14. Create a new object called *scope~*
15. Connect the first outlet of *gain~* to the first inlet of *scope~*
16. Open the *Inspector* for *scope~* and change the *Calccount - samples per pixel* to *20* (this changes the samples used to graphically represent the wave and makes the wave oscillation look smoother)

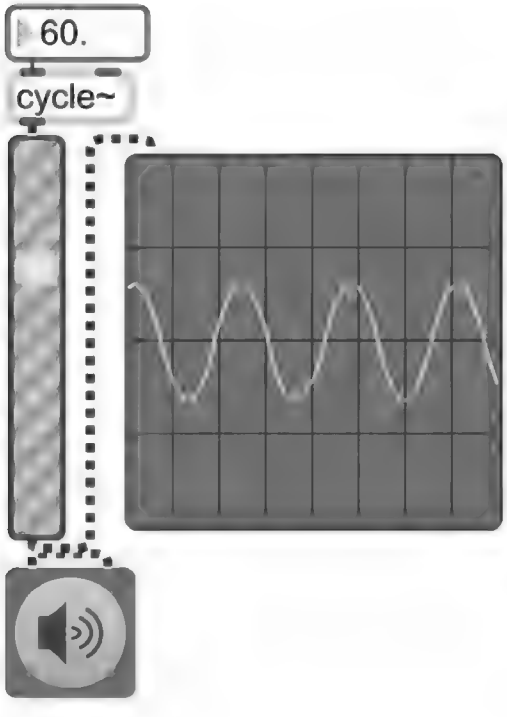


FIGURE 12.8
sine displayed in scope~ | sine_stuff.maxpat

17. Lock your patch and turn on the *ezdac~*

Notice that the wave is represented continuously in *scope~*; the more gain you give the wave through *gain~* the higher the wave will be in the representation. *Scope~* can even represent multiple waves.

18. Turn off the *ezdac~*

Unlock your patch and

19. Copy and paste a copy of everything in your patch except for the *ezdac~*[2] and the *scope~*
20. Connect the first outlet of the newly created *gain~* to the first inlet of *scope~*
21. Connect the first outlet of the newly created *gain~* to both inlets of the *ezdac~* object

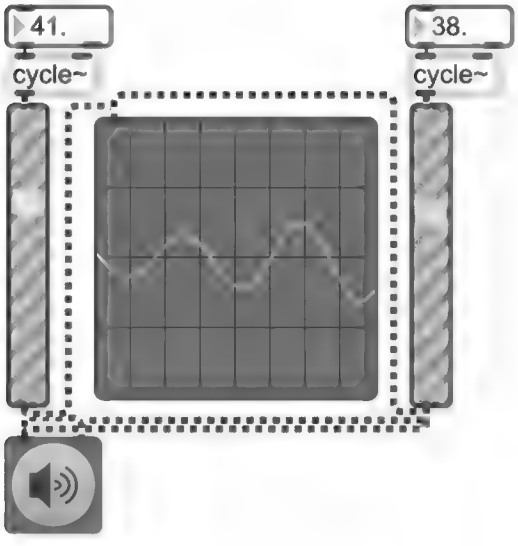


FIGURE 12.9
complex tone displayed in scope~ | sine_stuff .maxpat

22. Lock your patch, turn on *ezdac~*, and change the numbers to each *cycle~* object

Notice that the shape of the wave changes when combined with other sine waves. All MSP objects automatically sum multiple signals received in a single inlet. This creates what is known as a complex waveform. Click *File>Save* and save this patch as *sine_stuff.maxpat*.

Timbre

Here's a scenario: you're discussing some orchestration technique and a colleague mentions that the brass and woodwind sections are mimicking the harmonic series. You think to yourself, I remember hearing about the harmonic series, but I don't quite recall what it is or how it works. Is that sort of like the overtone series?

2. For the sake of clarity, you typically will only need one *ezdac~* or *ezadc~* object in your patch and should avoid creating multiple instances. If you intend to specify certain output/input channels with *dac~ or adc~*, multiple instances can be justified.

The answer to the last question is *yes*. The harmonic series is a set of frequencies (referred to as partials) in which overtones (harmonics) are built above some base tone (referred to as the fundamental frequency). The fundamental frequency is the first partial in a given harmonic series. The balance of overtones (other harmonics above the fundamental frequency) built from a fundamental frequency is an important factor in establishing the timbre of an instrument. So how do you build a harmonic series from a fundamental frequency?

The first overtone, which is the second harmonic (the first frequency above the fundamental) is an octave above the fundamental. Let's say that the fundamental frequency is *440* Hz, the *A* above middle C. The *A* one octave above *A 440* is at *880* Hz. So, to get the second harmonic of the series, you simply multiply the fundamental frequency by *2*. To get the third harmonic of the series, multiply by *3*, and so on. It's actually a really easy concept to grasp.

Let's build a harmonic series patch. With the *sine_stuff.maxpat* file still open, click *File>Save As* and save this patch as *harmonic_series.maxpat*. Unlock the patch and

23. Highlight the *scope~* object and delete it (we won't need it for this patch, but you can always add it back in later)
24. Create a new *flonum* box at the top of the patch
25. Create a new object called * with the argument *1.* (don't forget the period. This tells the object that the resulting number can be a floating-point number)
26. Connect the first outlet of the newly created *flonum* box to the first inlet of * *1.*
27. Connect the outlet of * *1.* to the inlet of the *flonum* box connected to the first *cycle~* object (Step 4)
28. Create a new object called * with the argument *2.*
29. Connect the first outlet of the newly created *flonum* box (Step 24) to the first inlet of * *2.*
30. Connect the outlet of * *2.* to the inlet of the *flonum* box connected to the second *cycle~* object (Step 19)

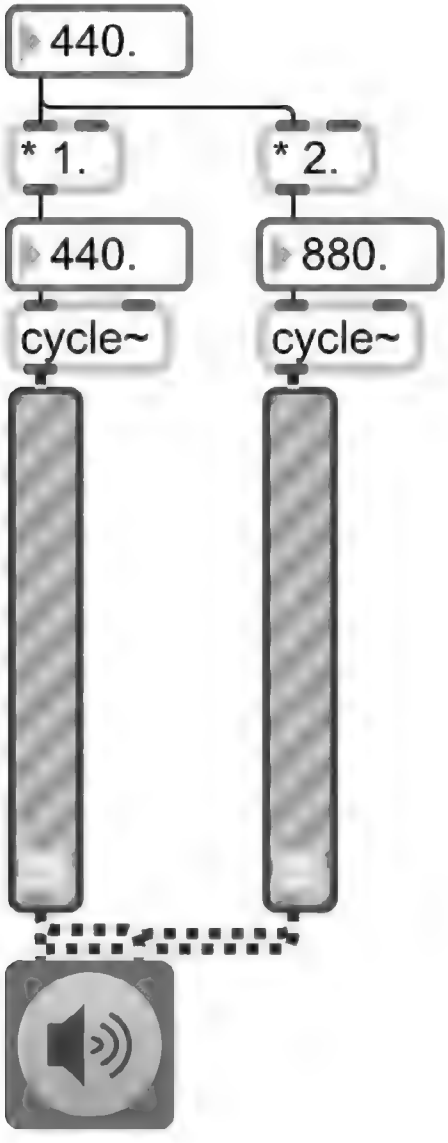


FIGURE 12.10
single floating-point number multiplied to obtain octave | harmonic _series.maxpat

31. Lock the patch and enter the number *440* into the topmost *flonum* box

In this patch, a single frequency value is entered into the topmost *flonum* box. It is then multiplied by *1.*, which is the same as just passing it right through to the next object, and also by *2.* If the first *cycle~* is thought of as generating the fundamental frequency of the harmonic series, then the second *cycle~* can be thought of as generating the first overtone, that is, the second harmonic exactly one octave above the fundamental frequency. To continue to build the harmonic series, simply copy the * object, the *flonum*, the *cycle~*, and the *gain~*, increasing the value of the argument for * with each new copy. Unlock your patch and

32. Copy the * *1.* object, the *flonum* receiving from *'s outlet, the *cycle~* receiving from *flonum*, and the *gain~* receiving from *cycle~* and paste 8 copies aligned horizontally

33. Connect the first outlet of topmost *flonum* object to the first inlet of each * object
34. Double click each newly copied * object and change the argument of each one to the increasing numbers *3.* through *10.* (don't forget the periods!)
35. Connect the first outlet of each *gain~* object to both inlets of the *ezdac~* object

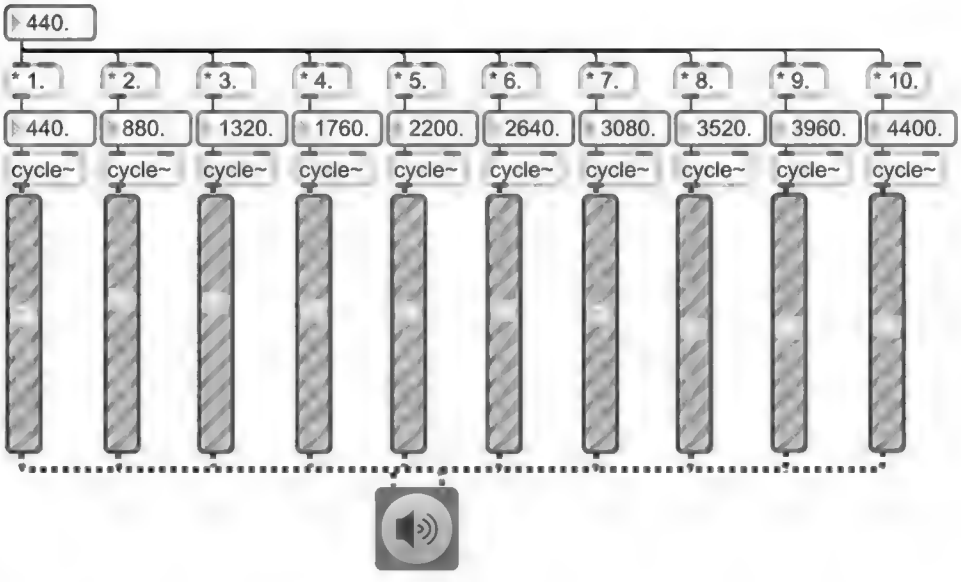


FIGURE 12.11

number multiplied many times to achieve harmonic series | harmonic_series .maxpat

36. Lock the patch and enter the number *440* into the topmost *flonum* box
37. Turn on the *ezdac~*
38. Slowly increase the slider on each *gain~* object
39. When you have heard the first 10 harmonics of the harmonic series built from *440* Hz, change the number in the topmost *flonum* box
40. Turn off the *ezdac~*

Discussing the harmonic series can be an insightful way to introduce concepts of timbre. It is also a great way to introduce the idea of *additive synthesis*, a type of synthesis in which a user has control over each harmonic and combines them to create new timbres or emulate existing ones. With control over enough sine waves, you could, in principle, construct a timbre that sounds like a Stradivarius (though that would take some time).

We can turn this patch into an additive synthesizer of sorts by adding in some MIDI input objects. The difficulty would be in converting MIDI notes to frequency values; there's some math involved. However, there is an object call

mtof (as in MIDI, *m*, *to* frequency, *f*) that can do the conversion for you.[3] Unlock your patch and

41. At the top of the patch, create a new object called *notein*
42. Create a new object beneath *notein* called *kslider*
43. Connect the first 2 outlets of *notein* to the 2 inlets of *kslider*
44. Create a *number* box beneath *kslider*
45. Connect the first outlet of *kslider* to the inlet of *number* box
46. Create a new object called *mtof* and place it beneath the newly created *number* box and above the top *flonum* box (notice that *mtof* does not have a ~ because it works with numbers and not audio)
47. Connect the outlet of the first outlet of the *number* box to the inlet of *mtof*
48. Connect the outlet of *mtof* to the inlet of the *flonum* box

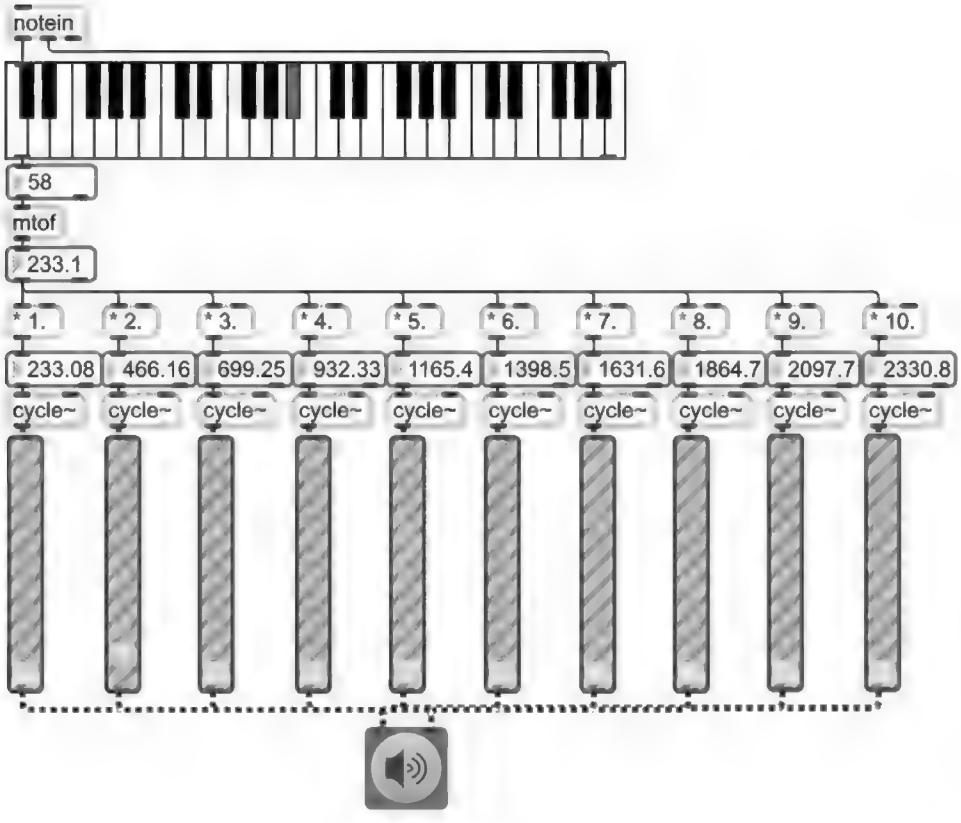


FIGURE 12.12

MIDI note converted to frequency by which a harmonic series is built | harmonic_series.maxpat

49. Lock the patch and play a note on your MIDI keyboard
50. Turn on the *ezdac~*

3. There's also an *ftom* object that converts frequencies to MIDI notes.

51. If they are not already at desired levels, slowly increase the slider on each *gain~* object
52. Try playing some other notes
53. Turn off the *ezdac~*

One of the interesting things about this patch is that if you play a note with only one harmonic present and slowly introduce the rest of the harmonics in the series by raising their gains, you may hear each frequency collectively as a tall chord, still being able to pick out each tone. However, if you leave the gain levels where they are and play sequential notes on the keyboard, you will likely hear all of the frequencies as a gestalt, in which the frequencies collectively constitute the timbre of a single pitch. Save your patch.

Synthesizer

In essence, we have created a nice monophonic synthesizer although there's no way to adjust the volume of this synthesizer. Our MIDI device outputs velocity values between *0* and *127*, and look at that—our *gain~* object can receive numbers between *0* and *127*. However, instead of just connecting the outlet of *kslider* to send velocity values to the inlet of each *gain~*, let's look at another way of doing it that gives us control over the attack and release of the note.

An object called *line* can allow you to glide up or down to a number over a specified amount of time. For instance, if you wanted to go from the number *21* to *100* over the course of 4 seconds (4,000 milliseconds), you can send the *line* object a *message* containing *21, 100 4000*. If you send *line* a *message* with two numbers in it, *line* will go from whatever number it received to the first number in your *message* over the amount of time indicated by the last number of your *message*. In other words, if *line* last received the number *100*, and we sent it a *message* containing *817 1500*, *line* would go from *100* to *817* in *1500* milliseconds. The plane of numbers between the starting and ending number is referred to as a "ramp."

54. Create a new object called *line*
55. Create a new object called *pack* above *line* with the arguments *0* and *500*
56. Connect the outlet of *pack* to the first inlet of *line*
57. Create a *number* box beneath *line*
58. Connect the first outlet of *line* to the inlet of the newly created *number* box
59. Create a *number* box beneath *kslider*'s rightmost outlet
60. Connect the last outlet of *kslider* to the inlet of this newly created *number* box

61. Connect the first outlet of this *number* box to the first inlet of *pack 0 500*
62. Connect the first outlet of the *number* box receiving from *line* to the first inlet of each of the 10 *gain~* objects

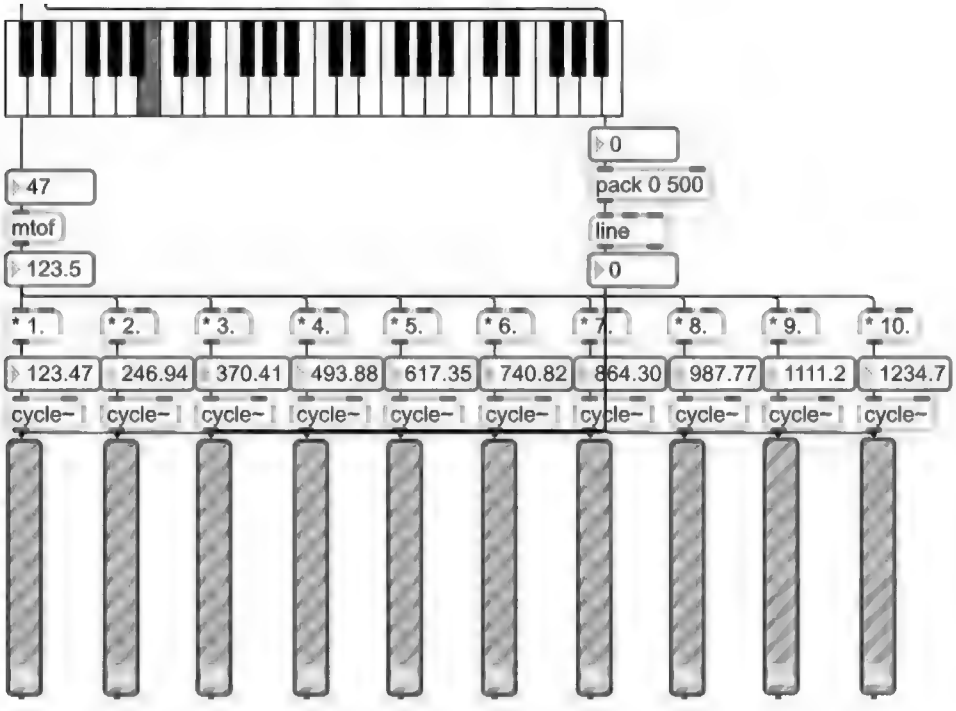


FIGURE 12.13
line object ramps gain~ value to match MIDI velocity in 500 ms | harmonic_series.maxpat

63. Lock the patch and turn on the *ezdac~*
64. Play a note on your MIDI keyboard
65. Notice that the *gain~* objects increase to the velocity you played over 500 milliseconds
66. Try playing some other notes
67. Turn off the *ezdac~*

We could have also used a *message* containing *$1 500* instead of *pack 0 500* to achieve the same result, but *pack* gives us the freedom to change *line*'s "ramp" speed. You may have noticed that if you play more than one note, the consecutive notes interfere with one another's velocity messages and, as a result, you could very well be holding a note down while the *gain~* objects slide down to *0*. That's not a good thing. However, an object called *ddg.mono* will help us here by handling the note on/off messages in the proper way. Unlock your patch and

68. Double click the *notein* object and replace the text with *ddg.mono*
69. Create a new object called *notein* above the *ddg.mono* object
70. Connect the first 2 outlets of *notein* to the 2 inlets of *ddg.mono,* respectively

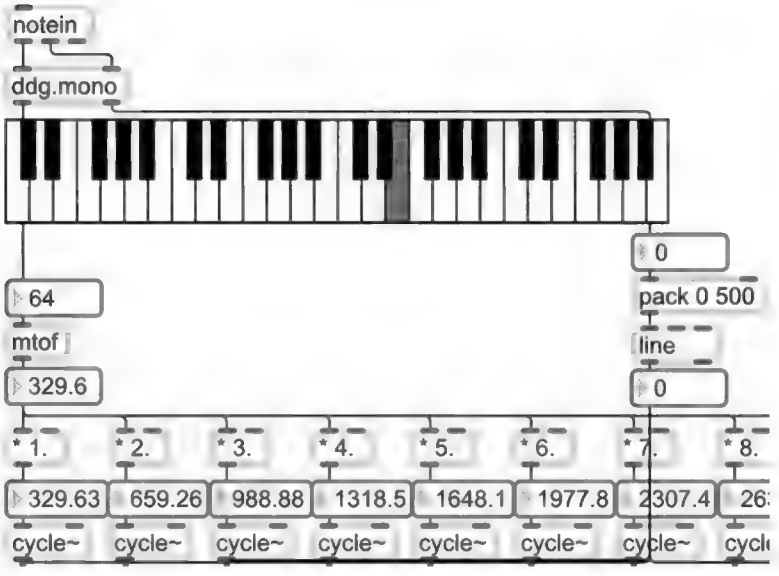


FIGURE 12.14
ddg.mono object handling MIDI messages | harmonic_series.maxpat

71. Lock the patch and Turn on the *ezdac~*
72. play a note on your MIDI keyboard

We now have a nice little patch that will play back each harmonic of the harmonic series starting from a fundamental played on a MIDI keyboard. Click *File>Save* to save this patch.

Now, even though we only intended to use this patch to demonstrate some aspects of timbre, we're really only a few steps away from turning this patch into a basic polyphonic synthesizer. Click *File>Save As* and this patch as *poly_synth_patch.maxpat*.

Synth Building

The patch we have right now is good for a single note synthesizer, but if we wanted to make a polyphonic synthesizer, we would need some way to divide up the MIDI notes as they come in and route them to different instances of this patch. You may remember that the *poly* object we discussed in Chapter 4 allowed us to route incoming MIDI notes to separate locations. Well, there is another object called *poly~* that will allow you to divide incoming MIDI notes and send each one to a separate instance of a signal processing patch for each MIDI note.

The *poly~* object is similar to a subpatcher in that the object takes the name of an existing patch as its argument. The *poly~* object is then used in a "main patch" and its content, the enclosed patch, is used inside of *poly~*. The enclosed patch typically does some sort of signal processing or, in our case, additive synthesis, and must contain a few objects that allow us to communicate with this patch.

Let's begin by formatting the *poly_synth_patch.maxpat* file you just created and make it suitable for use inside a *poly~* object. To get data into a *poly~* sub-

patcher, like this one, we use the object *in* with an argument specifying which number it is; for instance, *in 1*. Unlock your patch and

73. Create an object called *in* with argument *1* at the top of your patch above the *notein* object

We'll be receiving MIDI information in the "main patch" from a *midiin* object, so let's assume we'll be getting some pair of pitch and velocity values into this subpatcher through the *in* object. We'll want some way to separate the pitch and velocity values in order to use them with the rest of our patch, so we'll leave that task to an *unpack* object.

74. Double click on the *notein* object and change the text to *unpack* with the arguments *0 0*
75. Connect the outlet of *in 1* to the inlet of *unpack 0 0*

MIDI data will be sent from the "main patch" to this subpatcher through the *in* object where it will be unpacked and sent to the rest of the patch. You could very well, for that matter, make other *in* objects (*in 2, in 3,* etc.) to control different parameters inside the patch, such as the ramp time for *line*. We'll talk about this in greater detail later.

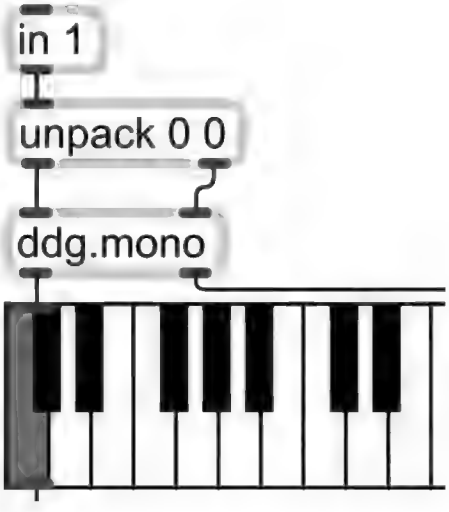


FIGURE 12.15
notein object replaced by unpack object | poly _synth_patch.maxpat

The next thing we need to do is get data out of this patch. The data we're interested in is the combined signal of each of these harmonics; in other words: everything that is currently connect to *ezdac~*. Because we're going to use this patch as a subpatcher, we want our signal to be sent from this patch, so we'll need to remove *ezdac~*. We'll replace *ezdac~* with a new object called *out~*.

In the same way that *in* gets data into the subpatcher, *out~* will get the signal out of this subpatcher. As you may have guessed, there is also a regular *out* object (without the ~) as well as an *in~* object. Remember, the ~ signifies that an audio signal is being sent through it as opposed to numbers and text.

76. Delete the *ezdac~* object
77. Create a new object called *out~* with the argument *1*
78. Connect the first outlet of each *gain~* object to the inlet of *out~ 1*

At this point, you're almost done. We simply need to include an object called *thispoly~* which handles the allocation of different MIDI voices to each instance of the embedded patch.

79. At the bottom of your patch, create an object called *thispoly~*

The *thispoly~* object also controls the muting of voices when they're not in use. Here's the only tricky part (which isn't really that tricky): you need to tell *thispoly~* when the subpatch is in use and when it should be muted. To do this, we'll send *thispoly~* two messages, one will be *toggle* to tell it to get busy working, and the second will be a *message* called *mute* with the argument either *1* (for muted) or *0* (for not muted).

80. Create a *toggle*
81. Connect the outlet of *toggle* to the inlet of *thispoly~*
82. Create a *message* box with the argument *mute $1*
83. Connect the first outlet of the *message mute $1* to the inlet of *thispoly~*
84. Create 4 *message* boxes containing the numbers *1, 0, 1,* and *0,* respectively (you will have 2 pairs of "1" and "0" *message* boxes)
85. Connect the outlet of the first 2 *message* boxes containing *1* and *0* to the inlet of the *toggle*
86. Connect the outlet of the last 2 *message* boxes containing *1* and *0* to the first inlet of the *message mute $1*
87. Create 2 *buttons* above the 2 pairs of *message* boxes containing 1's and 0's
88. Connect the outlet of the first *button* to the first inlet of the first 2 *message* boxes containing *1* and *0*
89. Connect the outlet of the second *button* to the first inlet of the last 2 *message* boxes containing *1* and *0*

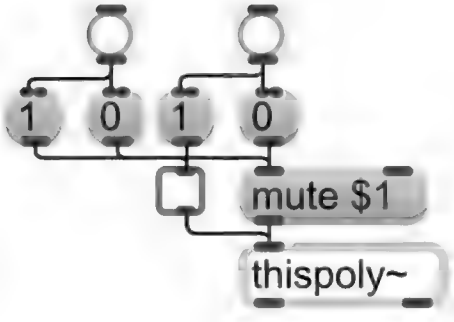


FIGURE 12.16

controlling the activity in this patch | poly_synth _patch.maxpat

We can now control two states of *thispoly~* with two *buttons*; when the first *button* is pressed, a *1* will be sent to the *message mute $1,* which will, in effect, be *mute 1,* causing *thispoly~* to be muted. At the same time, the *toggle* will receive a *0* which tells *thispoly~* that the patch is not busy doing anything. When the second *button* is pressed, the opposite takes place: *thispoly~* is told that the patch is busy doing something and is unmuted. The next logical step is to determine how we're going to trigger these two states.

For our purposes, let's use a *sel* object with the argument *0* so that when the *gain~* slider value is at *0,* a bang will be sent to the first *button.* When the velocity output of *ddg.mono* is anything other than *0,* let's bang the second *button.*

90. Create a new object called *sel* with the argument *0*
91. Connect the first outlet of *sel* to the inlet of the first *button*

92. Connect the second outlet of *sel* to the inlet of the second *button*
93. Connect the second outlet of the first *gain~* object to the first inlet of the *sel 0* object

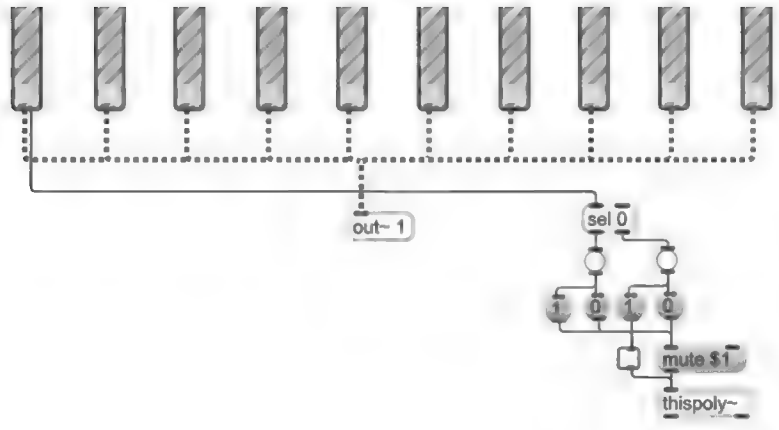


FIGURE 12.17
gain~ value of 0 mutes this patch | poly_synth_patch.maxpat

94. Save the changes to this patch and close it
95. Create a new patch and click *File>Save* and call it *poly_main_patch.maxpat* making sure that this patch is saved in the same folder as *poly_synth_patch.maxpat*
96. Create a new object called *poly~* with the arguments *poly_synth_patch* and *16*
97. Hold ⌘ (Mac) or ctrl (Windows) and double click the *poly~* object to open the embedded *poly_synth_patch* file in a new window

We have given *poly~* the name of a patch to load inside itself. We also gave it the argument *16* which means that *poly~* will maintain control of up to 16 instances (or virtual copies) of *poly_synth_patch*. In other words, for each note we send to *poly~*, a separate instance of the *poly_synth_patch* will be used to synthesize that note. Let's get some MIDI notes from our MIDI keyboard into *poly~* with the *midiin* object.

98. Create a new object called *midiin*
99. Create a new object called *midiparse*
100. Connect the first outlet of *midiin* to the inlet of *midiparse*

We used *midiin* because we want pitch and velocity pairs formatted in a single message which is what we get when we parse the raw MIDI data with *midiparse*. We could have also used *notein* and *pack* to pair the pitch and velocity messages together. The real task at hand is to prepend these pitch and velocity pairs with the message *midinote* so that *poly~* will know that the pitch and velocity numbers coming in are MIDI numbers.

101. Create a new object called *prepend* with the argument *midinote*
102. Connect the first outlet of *midiparse* to the inlet of *prepend midinote*
103. Connect the outlet of *prepend midinote* to the inlet of *poly~*

We need to add some objects in order to hear this synth.

104. Create a new object called *gain~*
105. Connect the outlet of *poly~* to the first inlet of *gain~*
106. Create a new object called *ezdac~*
107. Connect the first outlet of *gain~* to both inlets of *ezdac~*

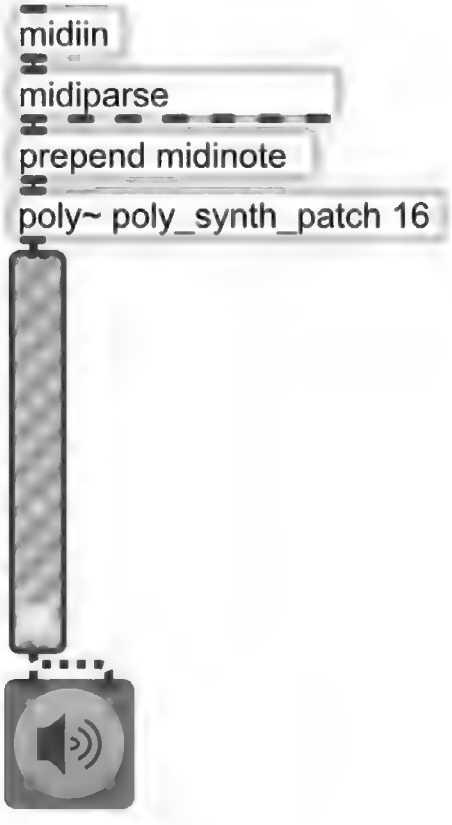


FIGURE 12.18

poly~ object contains poly_synth_patch.maxpat and sends it MIDI data | poly_main_patch.maxpat

108. Lock your patch, turn on the *ezdac~*, raise the *gain~* slider half way, and play a 3-note chord on your MIDI keyboard

Notice that the patch is, in fact, polyphonic. Well done! The *gain~* here acts as a main volume control for the *gain~*'s inside the *poly~* object similarly to the way a mixing board has gain knobs for individual channels, groups, and master output.

There are a few tweaks we can make to the patch at this point. For starters, if you send the *message open 1* to the *poly~* object, it will open the first instance of the embedded patch. Unlock your patch and

109. Create 3 *message* boxes containing the message *open 1*, *open 2*, and *open 3*, respectively
110. Connect the outlet of each *message* box to the inlet of *poly~*

Next, we should send the *message steal 1* to *poly~* to allow "voice stealing" so that if more than 16 notes are held down, the oldest notes will be dropped in place of newer notes.

111. Create a new *message* box containing the text *steal 1*
112. Connect the outlet of this *message* box to the inlet of *poly~*
113. Create a new object called *loadbang*
114. Connect the outlet of *loadbang* to the first inlet of the *message steal 1*

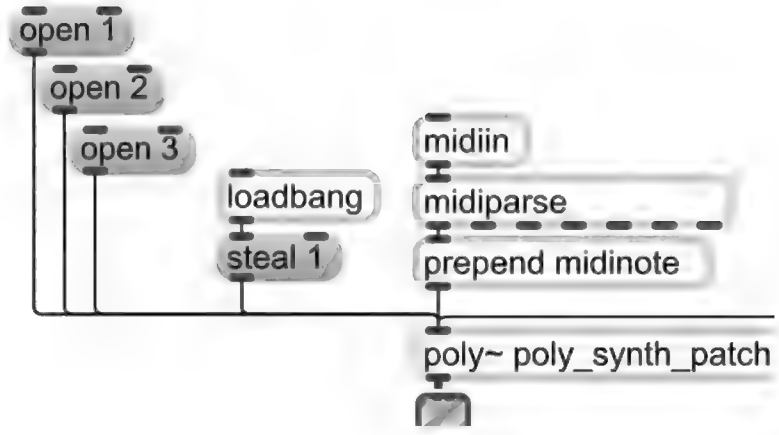


FIGURE 12.19
additional options set for poly~ | poly_main_patch .maxpat

115. Save the *poly_main_patch.maxpat* patch and close it.

Let's re-open *poly_synth_patch.maxpat* and add the ability to control the *line* speed.

116. Open *poly_synth_patch.maxpat* (Note: you can click *File>Open Recent* to see a list of recently opened files)
117. Create a new object at the top right of the patch called *in* with the argument *2*
118. Connect the outlet of *in 2* to the second inlet of *pack 0 500*
119. Save and close the patch
120. Open *poly_main_patch.maxpat*

Notice that there is now a second inlet for *poly~* as a result of adding the *in 2* object to its embedded patch. This inlet connects directly with the *pack* object to control the amount of time *line* will allot for the ramp from one velocity value to another. However, to use this inlet for each instance of the embedded patch in *poly~*, we need to supply a *target* message so that the data will go to the

appropriate patch. For example, if we just wanted the data to go to the second instance of the embedded *poly_synth_patch*, we would send the message *target 2* along with the data to *poly~*'s second inlet. Since we want the data going to *poly~*'s second inlet to apply to all instances of the embedded *poly_synth_patch*, we'll use the message *target 0*. We'll use a *trigger* object to execute this. Unlock the patch and

121. Create a new *number* box above *poly~*'s rightmost inlet
122. Create a new object called *t* with the arguments *i* and *b*
123. Create a *message* box containing the text *target 0*
124. Connect the last outlet of *t i b* (sends out a bang) to the first inlet of the *message target 0*
125. Connect the first outlet of *t i b* to the second inlet of *poly~*
126. Connect the outlet of *message target 0* to the first inlet of *poly~*

Save and lock your patch. Now, you can change the ramp time *line* uses from the default *500* milliseconds to anything you'd like (although I would recommend not going much lower than *30* milliseconds or else you may start to hear an unnatural popping in your speakers caused from turning the *gain~* values up while the waves are not at a "zero-crossing" point.

Congratulations, the synthesizer works! I wouldn't rush out to a flea market to sell your old reliable synthesizer in place of what you just created, but as a first software synth patch, this is a pretty good one. As you continue to learn about the MSP objects, you may want to revisit this patch and use it as a starting point for creating more synth sounds. Close this patch.

Keep in mind that there are many different aspects of signal processing and synthesis techniques that you can learn, but this book does not devote much time to explaining these concepts. If you do not know about signal processing and synthesis techniques, you may find it helpful to read about these techniques first since incorporating them into Max in the manner we just learned is the easy part. The MSP tutorials provide a number of examples of these techniques and how to implement them in Max with some explanation of the theory. As a result, some of the tutorials may not make much sense if you don't roughly know what is going on signal-wise to achieve the sonic result. If this is of interest to you, you may want to read *The Theory and Technique of Electronic Music* (Puckette, 2007) by Miller Puckette, the original author of Max.

Remember

- For information on the sound card connected to your computer and how its inputs and outputs are organized in Max, choose *Options>DSP Status* from the top menu. You can also get to the DSP Status by double clicking the *adc~* object or by sending *adc~* the *message open*.

- If you have trouble getting audio to play back in Max or simply want to test your audio, you may choose *Audiotester* from the *Extras* menu at the top menu.
- You only need one *ezdac~* object and one *adc~* object per patch.
- All MSP objects automatically use the sum of the signals received in a single inlet.

New Objects Learned:

- *adc~*
- *meter~*
- *gain~*
- *dac~*
- *ezadc~*
- *ezdac~*
- *mtof*
- *line*
- *ddg.mono*
- *poly~*
- *in*
- *in~*
- *out*
- *out~*
- *thispoly~*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select MSP Tutorials from the Help menu and read
  - MSP Tutorial 1—Test Tone
  - MSP Tutorial 2—Adjustable Oscillator
  - MSP Tutorial 3—Wavetable Oscillator
  - MSP Tutorial 4—Routing Signals
  - MSP Tutorial 5—Turning Signals On and Off
  - MSP Tutorial 6—A Review of Fundamentals
  - MSP Tutorial 7—Additive Synthesis

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.

- Create a patch that will aid in orchestration by showing the harmonic series (at least the first 5 notes) for a given fundamental note played on a MIDI keyboard. For practice, convert the fundamental to a frequency and build the harmonic series from that floating-point number. Convert the frequencies back to MIDI pitches and display them all on an *nslider* or *kslider*.

13

Audio Playback and Pitch Tracking

In this chapter, we will look at some of the ways that you can play back and record sound files. As you know, Max lets you design the way you control the variables in your patch. We will apply these design concepts to the ways we control the playback of recorded sound. We will also look at some ways to track the pitch of analog audio and convert it into MIDI numbers.

By the end of this chapter, you will have written a program that allows you to play back sound files using a computer keyboard as a control interface as well as a program that tracks the pitch you're singing from a microphone and automatically harmonizes in real time.

Playback

We will create a simple patch that plays back some prerecorded files I have prepared. Please locate the8 "*.aif*" audio files located in the *Chapter 13 Examples* folder.

1. Copy these 8 audio files to a new folder somewhere on your computer
2. In Max, create a new patch
3. Click *File>Save As* and save the patch as *playing_sounds.maxpat* in the same folder where you put these 8 audio files. There should be 9 files total in the folder (8 audio and 1 Max patch)

4. Close the patch *playing_sounds.maxpat*
5. Re-open *playing_sounds.maxpat* (the audio files will now be in the search path of the Max patch)

We can play back the prerecorded audio files we just copied using an object called *sfplay~*. The *sfplay~* object takes an argument to specify how many channels of audio you would like the object to handle. For instance, if you are loading a stereo (two channel) file, you can specify the argument *2*. Loading a sound file is easy: simply send the *sfplay~* object the *message open*. Playing back the sound is just as easy: send it a *1* or a *0* from a *toggle*. Let's build a patch that plays back these files. Unlock your patch and

6. Create a new object called *sfplay~*
7. Create a *message* box containing the text *open*
8. Connect the outlet of the *message open* to the first inlet of *sfplay~*
9. Create a *toggle*
10. Connect the outlet of *toggle* to the first inlet of *sfplay~*

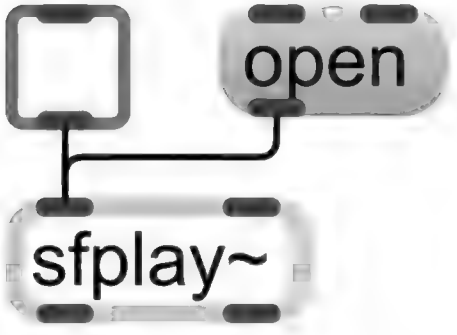


FIGURE 13.1
messages needed to open and play a sound file through sfplay~ | playing_sounds.maxpat

If you lock your patch and click on the *open* message, a dialog box will appear where you can find an audio file to load. The *sfplay~* object doesn't work with some formats like "*.mp3*" files. For now, we will stick with the "*.aif*" files that we have loaded in the same folder as this patch. In fact, since they are in the patch's search path, we don't need to browse for them using the dialog box that appeared when you clicked *open*. We can simply append the filename to the *open message*. With the patch unlocked

11. Create a new *message* box containing the text *open drums.aif*
12. Connect the outlet of the *message open drums.aif* to the first inlet of *sfplay~*

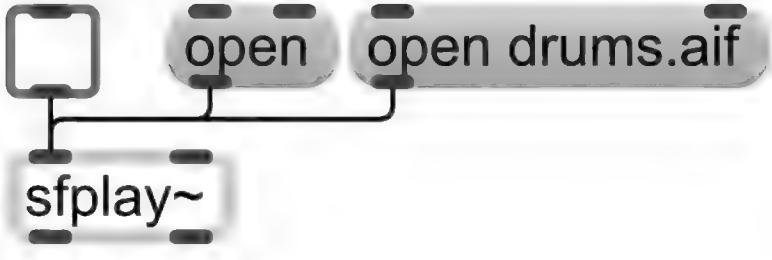


FIGURE 13.2
message containing specific file name loads a file in the Max search path | playing_sounds.maxpat

Let's add in the rest of the required components so we can hear this sound file played back.

13. Create a new object called *gain~*
14. Connect the first outlet of *sfplay~* to the first inlet of *gain~*
15. Create an object called *ezdac~*
16. Connect the first outlet of *gain~* to both inlets of *ezdac~*

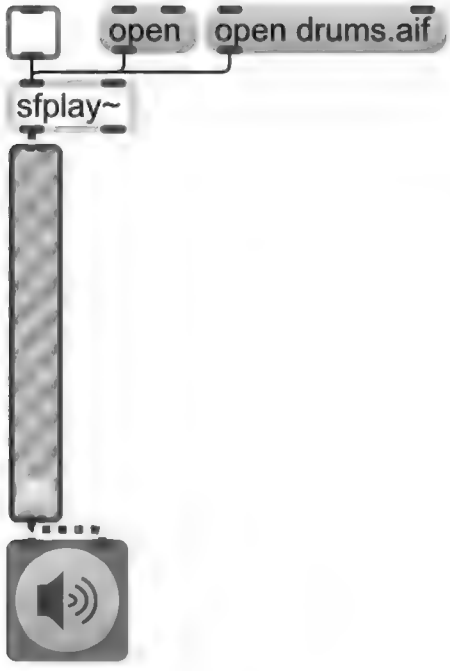


FIGURE 13.3
sound file playback is given gain control and sent to audio output | playing_sounds.maxpat

17. Lock the patch and click on the *message open drums.aif*
18. Turn on the *ezdac~*
19. Raise the *gain~* object about half way
20. Click on the *toggle* to send a *1* to the *sfplay~* (this will cause the file *drums.aif*) to play
21. Turn off the *ezdac~* when you have finished

Sending the number *1* to *sfplay~* as a *message* will allow you to replay the file from the beginning whenever it is pressed. In addition, sending the *message seek* followed by a number of milliseconds will start the file playback from that number of milliseconds into the file. For instance, the *message seek 500* will start playback from *500* milliseconds into the file; the *message seek 500 600* will start playback at *500* milliseconds into the file and stop at *600* milliseconds into the file. Unlock your patch and

22. Create a *message* box containing the number *1*
23. Create a *message* box containing the word *seek* and the numbers *500* and *600*
24. Create a *message* box containing the word *seek* and the numbers *0* and *200*
25. Connect the outlet of each of the 3 newly created *message* boxes to the first inlet of *sfplay~*

26. Lock your patch and turn on the *ezdac~*
27. Press the *message* boxes you just created
28. Turn off the *ezdac~* when you have finished

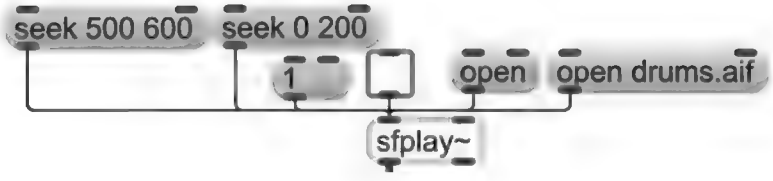


FIGURE 13.4
seek messages specify a selection of the file for playing | playing_sounds .maxpat

When I introduce *sfplay~* in my courses, I am shortly thereafter introduced to a number of student patches that play back recorded excerpts from various bands, TV shows, and celebrity quotes. Among my favorite student projects of this nature were the ASCII keyboard that played music excerpts by *Daft Punk* and the MIDI keyboard that played popular movie quotes by Samuel L. Jackson. Creating a patch similar to these is as easy as connecting a *key* to a *sel* and banging a *1* message. Imagine giving others a program with a palette of prerecorded sounds, perhaps some recordings of themselves speaking or singing, and allowing them to control the order in which these sounds appear just by pressing keys on their computer keyboard. Unlock your patch and

29. Create a new object called *key*
30. Create a new object called *sel* with the argument *32*
31. Connect the first outlet of *key* to the first inlet of *sel 32*
32. Create a *button*
33. Connect the first outlet of *sel 32* to the inlet of the *button* (the key *32* is the space bar)
34. Connect the outlet of the *button* to the first inlet of the *message 1*

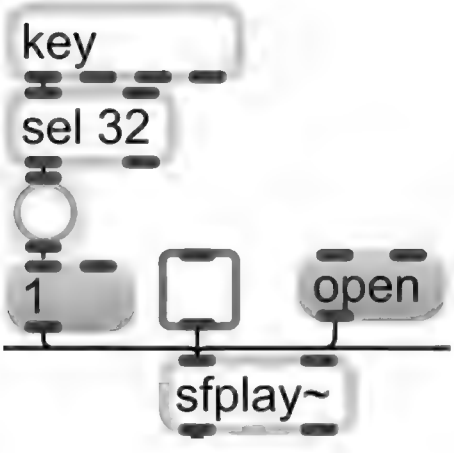


FIGURE 13.5
space bar plays sound file | playing_sounds.maxpat

The space bar will now play the sound file. If you are interested in looping the file continuously, you can use the *message loop 1* to turn looping on and *loop 0* to turn it off. This feature is especially useful for longer "pad-like" or "textural" sound beds where a user can layer multiple sound files on top of each other and simply control the volume of each sound by adjusting the *gain~* slider.

35. Create a new *message* containing the text *loop $1*
36. Connect the outlet of the *message loop $1* to the first inlet of *sfplay~*

37. Create a *toggle* above the *message loop $1*
38. Connect the first outlet of *toggle* to the first inlet of the *message loop $1*

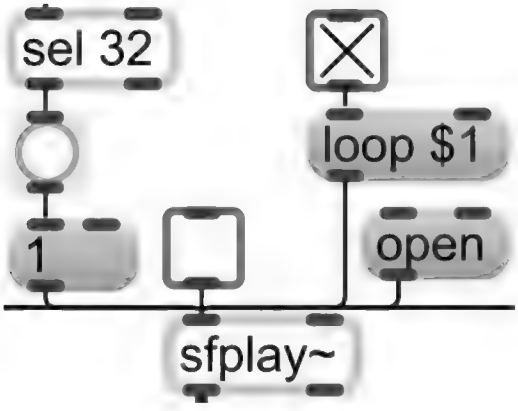


FIGURE 13.6
loop1 message repeats sound file playback continually | playing _sounds.maxpat

39. Lock your patch and click on the *toggle* above the *message loop $1* to enable looping
40. Turn on the *ezdac~*
41. Press the space bar to start looped playback
42. When you have finished, set the *toggle* connected directly to *sfplay~* to its *0* (off) setting or simply uncheck the *toggle* connected to the *message loop $1* and the file will stop looping when it has finished playing
43. Turn off the *ezdac~* when you have finished

Let's get some more sound files in this patch. Unlock your patch and

44. Copy everything in the patch except for the *ezdac~* and paste a copy to the right of the existing objects
45. Connect the first outlet of the newly copied *gain~* to both inlets of the *ezdac~*
46. Double click the *message* box containing the text *open drums.aif* and change the text to *open bass_guitar.aif* (Note that the underscore "_" is used so that there are no spaces in the filename. It's a good idea to follow this same practice when possible)
47. Double click the newly copied *sel 32* object and change its argument from *32* to *9* (the key *9* is the tab key)

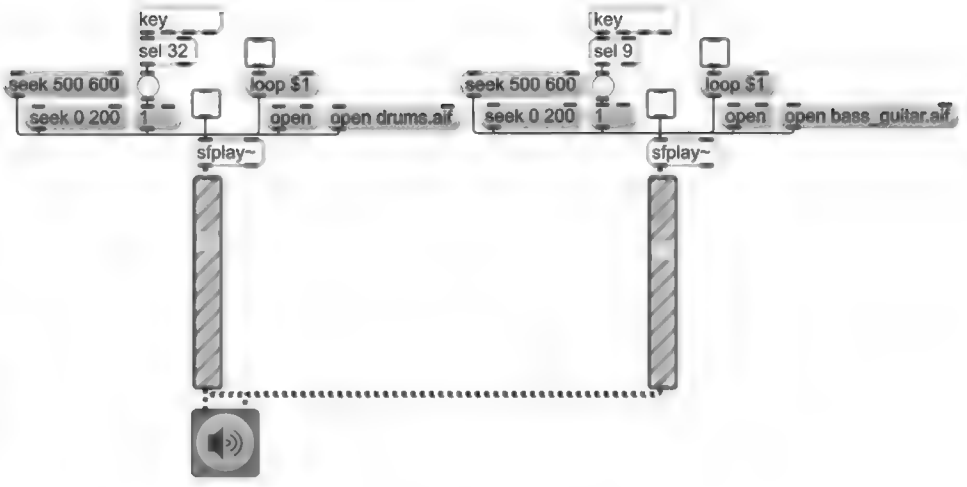


FIGURE 13.7

two different sfplay~ entities controlled by two different keys | playing _sounds.maxpat

48. Lock the patch and click on the *open bass_guitar.aif* message
49. Enable looping for both *sfplay~* objects by clicking the *toggles* connected to *loop $1*
50. Turn on the *ezdac~* and raise both *gain~* objects to about half way
51. Press the space bar and the tab key to play back both files (Note: you may still use the *toggles* connected to *sfplay~* to control playback in addition to the key commands we've created with *key* and *sel*)
52. Turn off the *ezdac~* when you have finished

It would be great to record this performance into an audio file. We can do this using the object *sfrecord~*. Like *sfplay~*, the *sfrecord~* object takes an argument to specify the number of channels of audio it will record. To record a stereo file, give *sfrecord~* the argument *2*. Also similar to *sfplay~*, to start recording with *sfrecord~*, first send the object the *message open* and specify a file name in the dialog box that opens. Next, send *sfrecord~* a *toggle* to begin and end the recording.

We want to record the audio coming from both of the *gain~* objects. Unlock your patch and

53. Create a new object called *sfrecord~* with the argument *2*
54. Connect the first outlet of each *gain~* object to both inlets of *sfrecord~* (there should be 2 audio patch cords connected to each of *sfrecord~*'s inlets)
55. Create a *message* box containing the text *open*
56. Connect the first outlet of *message open* to the first inlet of *sfrecord~*
57. Create a *toggle*
58. Connect the first outlet of *toggle* to the first inlet of *sfrecord~*

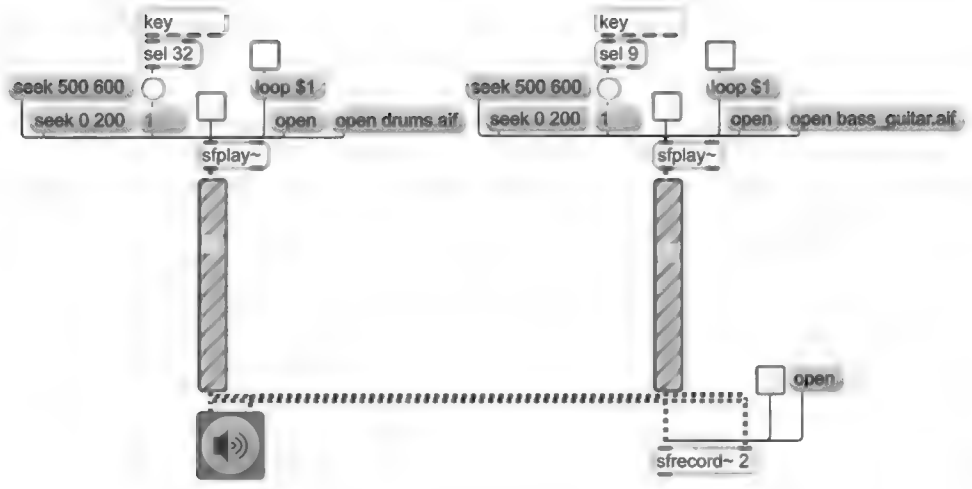


FIGURE 13.8

sfrecord~ allows performance to be recorded to a file | playing_sounds.maxpat

To record a performance:

59. Lock the patch and click on the *message open* connected to *sfrecord~* and specify a name for the file in the dialog box (such as *my_recording*) and click *Save*
60. Turn on the ezdac~
61. Click the *toggle* connected to *sfrecord~* to begin recording
62. Play back the files by using the space bar and tab key. Adjust the *gain~* sliders as you see fit
63. When you have finished performing, click the *toggle* connected to *sfrecord~* again to end recording
64. Turn off the *ezdac~*

Your recorded file will be ready for playback in whatever location on your computer you saved it (Step 59). As you know, you can use *loadbang* objects to load default variables such as filenames for your patch so that files are loaded when the patch is opened. For now, click *File>Save* and save and close this patch.

Pitch to MIDI Tracking

Converting a frequency number into a MIDI number is an easy task with *ftom*. However, converting an analog sound from your microphone into a MIDI number is a much more complicated process. A process known as a fast Fourier transform (FFT) must be performed on the audio signal to break the signal down into discrete frequencies. From each of the frequencies present, you would then have to identify the fundamental frequency of the complex sound and convert it to a MIDI pitch. Not an easy task.

Although there are several objects to aid in performing an FFT on a signal (see *fft~* and all related objects), a discussion of FFTs is beyond the scope of this

book. Since there is presently no "pitch tracking" object included in Max, we will look to one of the many external objects available for doing such a conversion.

Among the more popular objects that have existed for this type of pitch tracking are the *fiddle~* and *sigmund~* objects by Miller Puckette[1] (Puckette), and the *pitch~* and *analyzer~* objects by Tristan Jehan[2] (Jehan) which are based on the *fiddle~* object by Puckette. For this demonstration, we will use the *pitch~* object because it runs on both Windows and Mac platforms. Of course, there are other similar "pitch tracking" objects that you should try which may better suit your needs or run better for your system. For our purposes, we will be focusing more on implementation of these types of objects than the specific objects themselves. These types of "pitch tracking" objects typically give you the same information: a fundamental frequency value and/or the MIDI equivalent. It is their accuracy and efficiency in detecting the pitch that tend to differ.

When you installed the EAMIR SDK, the installer copied the *pitch~* object into the Max search path. For Windows users, it also copied a small driver file, fftw3.dll, into your Windows system directory.[3] Let's look at a patch I've prepared that illustrates one way that pitch tracking objects can be implemented in Max.

1. Click on *Extras>EAMIR* from the top menu to view the main menu of the *EAMIR SDK*
2. In the *umenu* labeled *Examples*, click the item called *EAMIR _audio_to_MIDI.maxpat*

The patch will load with a few *bpatcher* objects in it; nothing you haven't seen before. The *bpatcher* labeled "Audio Input" contains an *ezadc~* object and a few *umenu* objects for selecting from the different audio devices on your computer. The *bpatcher* labeled "Audio Output" contains an *ezdac~* object and similar *umenus*. The two *bpatchers* in the center of the patch have *gain~* objects. There are two *message* boxes connected to them containing the numbers *0* and *128* so that you can quickly turn a *gain~* slider completely off or to "unity gain" by clicking one of these *message* boxes. Again, these *bpatcher* objects don't contain anything we haven't already discussed. The second *gain~ bpatcher* is sent directly to the "Audio Output" *bpatcher,* while the first *gain~ bpatcher* is sent to the "Pitch to MIDI" *bpatcher*.

The *bpatcher* labeled "Pitch to MIDI" contains the *pitch~* object.

1. *sigmund~* by Miller Puckette; MSP port by Miller Puckette, Cort Lippe, and Ted Apel; *fiddle~* by Miller Puckette; MSP port by Ted Apel and David Zicarelli.

2. *pitch~* and *analyzer~* by Tristan Jehan, Windows ports by Paul Hill, all Universal Binary ports and bug fixes by Michael Zbyszynski.

3. If you did a manual install of these objects on Windows, be sure to copy the file *fftw3.dll* from EAMIR _SDK\EAMIR_3rd_Party_externals\tristan pitch in the Max application folder to the Windows system folder (C:\Windows\SysWOW64) for 64-bit OS or (C:\Windows\System32) for 32-bit OS.

3. Ctrl+click (Mac) or right click (Windows) this *bpatcher* and select *Object>Open Original "EAMIR_pitch_to_MIDI.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching mode.

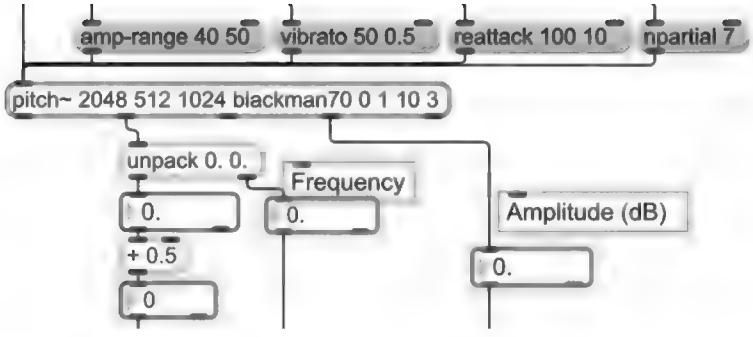


FIGURE 13.9
data received from pitch tracking object | EAMIR _pitch_to_MIDI.maxpat

This particular object has a number of arguments that can be used to tweak the pitch tracking results. For example, the *message npartial 7* refers to the number of partials above the fundamental frequency and their weighting in determining the fundamental. The *pitch~* object works well with the default values given in the Help file, but as you use this or similar pitch tracking objects, you may discover that other recommended settings work better for your specific system and needs.

Regardless of the setting, this object outputs what we want: a MIDI pitch and its related frequency value. As you can see, an *unpack* was used to parse the MIDI pitch and frequency numbers. Floating-point values were given. There are, however, only whole numbers in MIDI, so the first outlet of *unpack* is sent to a + *.5* to average the value. If the MIDI value given is over *.5*, adding *.5* to it will push it over to the next whole number. When the floating-point number is sent to a *number* box, the decimal point value is dropped. The MIDI note is sent out of the first outlet of this *bpatcher.* Close this patch.

In the *EAMIR_audio_to_MIDI* patch, the MIDI output number is then sent to a *change* object that, as you'll recall, only lets a number pass through if it is different from the preceding one. If it is different, *change* sends out the new number which causes a bang to stop sustaining any currently held MIDI notes and pairs the MIDI pitch with a velocity value through a *pack* object. The pitch and velocity pair is then unpacked to the pitch and velocity inlets of the "MIDI out" *bpatcher.*

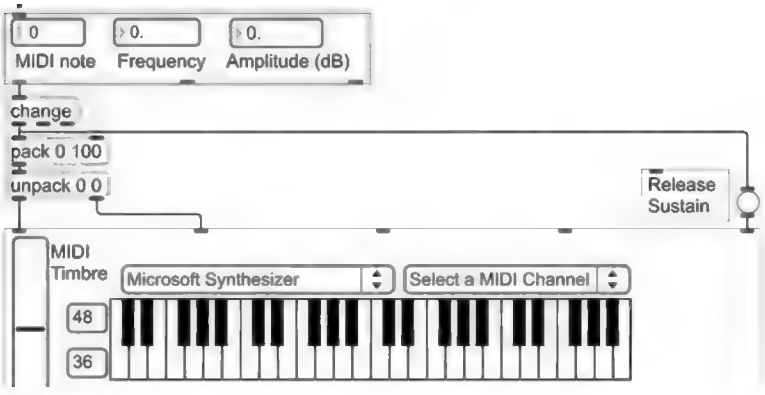


FIGURE 13.10

pitch tracking bpatcher connects to MIDI output bpatcher | EAMIR_audio _to_MIDI.maxpat

4. Lock the patch and turn on the *ezadc~* object (looks like a microphone) in the "Audio Input" *bpatcher* (Notice that the *ezdac~* object also turns on once the audio is initialized)
5. Sing into your computer's microphone and watch the MIDI note follow your pitch (You may also wish to try this with an electric guitar connected directly to your computer's sound card, though you may need to change the input source from "internal mic" to "line in." You may have to adjust the gain level for this to work properly)

Let's combine this patch with one of the harmonization techniques we've already discussed, so that when a user sings, his or her voice is harmonized. Click *File>Save As* and save this patch as *pitch_to_chords.maxpat*.

6. Delete the "MIDI Output" *bpatcher* as we'll be creating our own
7. Click on *Extras>Modal_Object_Library* from the top menu to view the main menu of the *Modal Object Library*
8. Click on the *message* box containing the text *modal_pc_match* to open the Help file for this abstraction
9. Unlock the Help file and copy the *loadmess 0,* the two *umenus*, the *modal_change* object, and *modal_pc_match*
10. Close the Help file
11. Paste the copied objects into the *pitch_to_chords* patch

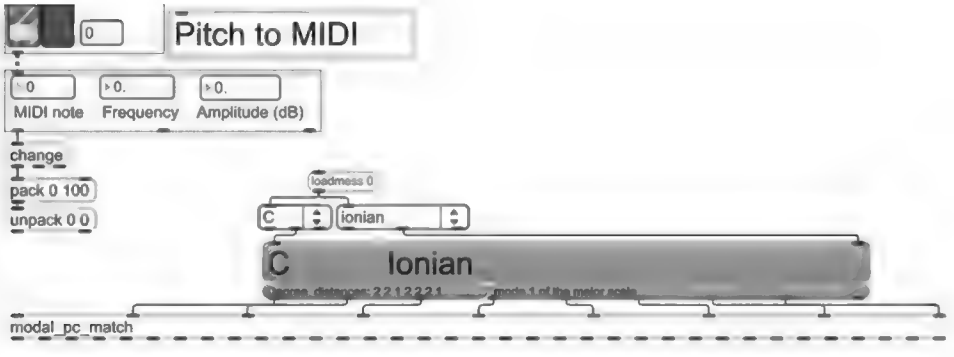


FIGURE 13.11

pitch tracking bpatcher modal_pc_match | pitch _to_chords.maxpat

As you recall, the *modal_pc_match* abstraction was a shortcut we used to compare an incoming pitch to one from a scale. We can do the same thing with *select* or *match* objects, but this way will be faster. The reason we copied directly from the Help file for *modal_pc_match* is that the *modal_change* object and other objects are already connected to each other. At this point, we simply need to send our tracked MIDI pitch to *modal_pc_match*.

12. Connect the first outlet of *unpack 0 0* to the first inlet of *modal_pc_match*

The *modal_pc_match* abstraction will send a bang from its first outlet when scale degree one of the selected tonic and mode is received. It will send a bang from its second outlet when scale degree two is received, and so on. We will later connect *message* boxes with chord names/functions to these outlets, which will ultimately harmonize our note. For now, let's add some MIDI output objects and a *modal_triad* object to receive the chord messages.

13. Create a new object called *modal_triad*
14. Connect each of the outlets of *modal_change* to the third through tenth inlets of *modal_triad,* respectively (these transmit the pitch from the selected scale to *modal_triad* where chords can be built)
15. Create a new object + beneath *modal_triad* with the argument *48*
16. Connect the first 3 outlets of *modal_triad* to the first inlet of + *48*
17. Create a new object beneath + *48* called *makenote* with the arguments *100* and *500*
18. Connect the outlet of + *48* to the first inlet of *makenote 100 500*
19. Create a new object called *noteout*
20. Connect both outlets of *makenote 100 500* to the first 2 inlets of *noteout,* respectively
21. Hold down the ⌘ (Mac) or ctrl (Windows) key and double click the *loadmess 0* object to reinitialize the *modal_change* object sending the notes of the scale to *modal_triad*

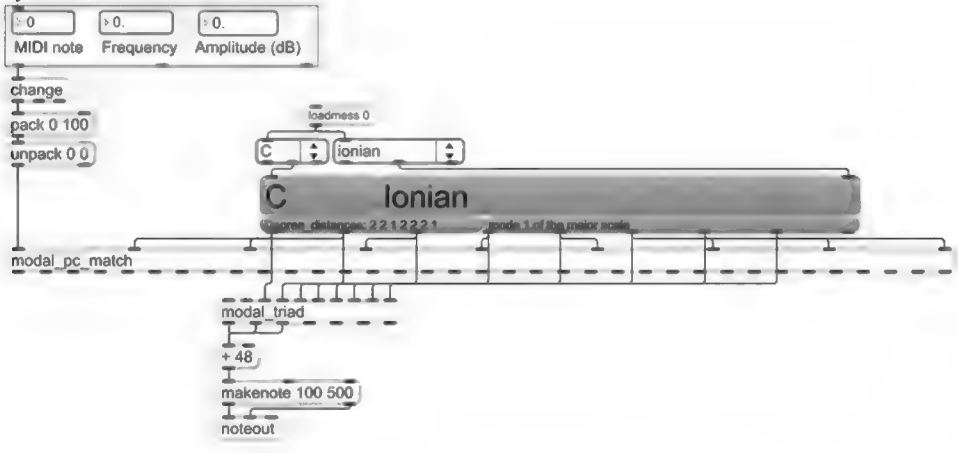


FIGURE 13.12

pitches are tracked and compared to a scale in preparation for harmonization | pitch_to_chords .maxpat

Now we reach the fun part: harmonization. As you know, the *modal_triad* object is just a shortcut object we've been using to quickly generate chords in a specified key. If we send it the *message 1*, it will give us the notes of the 1 chord in the specified key. We can create *message* boxes with chord names/functions and connect them to the inlet of *modal_triad*, which will allow us to use the bangs from *modal_pc_match* to trigger those chords when *modal_pc_match* receives a pitch from our pitch tracking portion of the patch. Let's begin with harmonizing the diatonic scale degrees:

22. Create 8 new *message* boxes containing the numbers *1–8*, respectively
23. Connect the first outlet of each of the 8 *message* boxes to the first inlet of *modal_triad*
24. Connect the first outlet of *modal_pc_match* to the first inlet of the message *1*, the second outlet to the first inlet of the message *2*, and so on until the first 8 outlets of *modal_pc_match* are connected to the first inlet of a *message* box

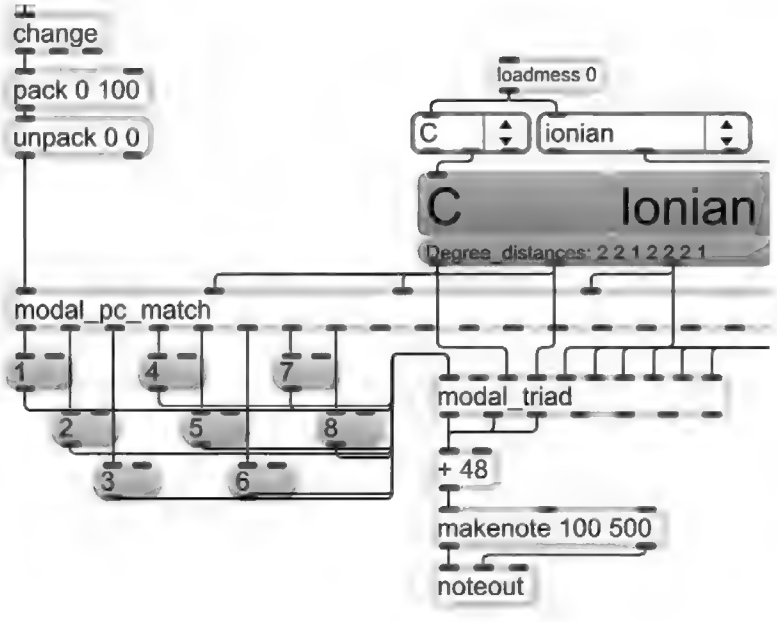


FIGURE 13.13

pitches are tracked, compared to a scale, and harmonized| pitch_to _chords.maxpat

If desired, you can use the remaining outlets of *modal_pc_match* to trigger chords to harmonize chromatic pitches when received. For this, I would recommend using some of the secondary dominant or other tonicizing chords that *modal_triad* understands (see the Help file for *modal_triad* for a complete list).

At this point, the patch will analyze a note sung into the microphone and, if it's a note from the selected scale, harmonize it with a chord. Be mindful of your closeness to the microphone and the *gain~* level when working with pitch tracking as a signal too loud or too soft could confound the pitch tracking ob-

ject. It's also a good idea to sing in tune; not just for the software, but for those around you. Save and close this patch.

New Objects Learned:

- *sfplay~*
- *sfrecord~*
- *pitch~* [external object]

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select MSP Tutorials from the Help menu and read
  - MSP Tutorial 13—Recording and Playback
  - MSP Tutorial 16—Record and Play Audio Files

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that uses a control interface of some sort (keyboard and mouse are acceptable) to play back audio files of some related nature and manipulate the gain values. Audio files can be anything: found sounds, recorded street noise, separate voices in a Bach fugue, recordings of yourself singing each note of the major scale in an alien voice, and so on. Make sure that the audio files are in the same folder as the patch you create so that they can be set to load automatically when the patch is opened. The user of this patch should not have to do anything except turn on the *ezdac~* and use the controls in order to hear sound.

14

Audio Buffers

In the last chapter, we discussed ways to play back prerecorded audio as well as ways to record audio. In this chapter, we will discuss how to record audio into a storage container called a buffer. When we record audio into a buffer, we can manipulate it in various ways in real time.

By the end of this chapter, you will be able to record a performance through your microphone and loop the recording while you make sonic changes to it somewhat like the "MIDI Looper" we made in Chapter 10. You will also learn how to make a polyphonic synthesizer that uses a single recording of your voice as the pitches.

All about Buffers

Buffers are storage containers for audio. Once audio is contained inside a buffer, the audio can be played back at a variable speed, looped, reversed, and have other types of manipulations performed on it. Buffers are great for "live looping" in which performers may record a musical idea into a buffer and then have the idea play repeatedly as they improvise or perform a second line on top of it. This technique is similar in function to the "MIDI looper" we made in Chapter 10 but uses actual audio instead of MIDI. Users may also decide to change aspects of the "looped" audio in the ways mentioned above while they are performing. We will begin our exploration of buffers by loading a prerecorded audio file into a buffer. Later on, we will allow our buffer to hold audio we record in real time.

To implement a buffer in Max, we need the *buffer~* object. We will give the *buffer~* object a name as its argument to distinguish it from other *buffer~* ob-

jects. Like other objects that take names as their argument such as *coll* and *table*, naming the *buffer~* allows other MSP objects with the same name argument to work with the same audio inside of the buffer. In a new patch

1. Create a new object called *buffer~* with the argument *mybuff* as a name
2. Hold ⌘ (Mac) or ctrl (Windows) and double click the *buffer~ mybuff* object to open the actual buffer window



FIGURE 14.1
buffer~ object with the name mybuff | all_about _buffers.maxpat

You will notice that the window that popped up looks like a big rectangle. This is the graphical representation of the buffer we named *mybuff*. At this point, the buffer has no defined size in milliseconds. If we loaded an audio file into this buffer by sending it a *message* box with the text *read*, the buffer would become the size of the audio we loaded into it. For the purpose of this introduction to buffers, we will supply a buffer size in milliseconds as a second argument for *buffer~*.

3. Double click the *buffer~ mybuff* object so that you can add the argument *4000* (Note: the object should read *buffer~ mybuff 4000 with spaces between both of the arguments*)
4. Create a new message box called *read* above the *buffer~* object
5. Connect the outlet of *read* to the first inlet of *buffer~*
6. Create a new message box called *read jongly.aif* above the *buffer~* object
7. Connect the outlet of *read jongly.aif* to the first inlet of *buffer~*

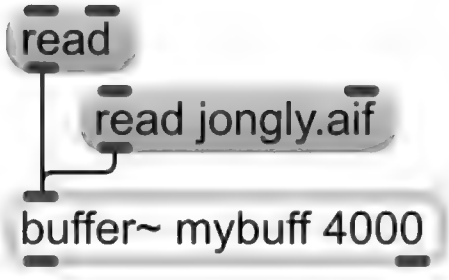


FIGURE 14.2
reading an audio file into a buffer | all_about _buffers.maxpat

The buffer size is now set to 4 seconds (4000 milliseconds). If we click the read message, we can load an audio file into the *buffer~* object and, as long as the file is not longer than 4 seconds, it will all fit within the buffer. To speed things up, we created a *message* called *read jongly.aif* which refers to an audio file that was installed on your computer when you installed Max. Since this file is in the Max search path, clicking on the *message read jongly.aif* will load that file, *jongly.aif*, into the *buffer~*.

8. Hold ⌘ (Mac) or ctrl (Windows) and click the *message read jongly.aif*
9. Hold ⌘ (Mac) or ctrl (Windows) and double click the *buffer~* object to open the buffer window

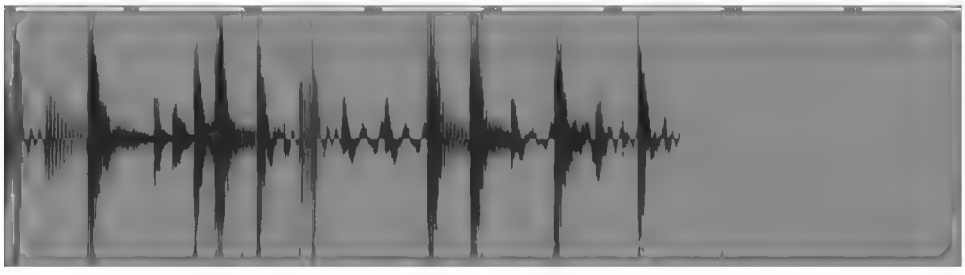


FIGURE 14.3

a 4-second buffer containing the audio file jongly.aif | all_about _buffers.maxpat

Notice that the *buffer~* window now has a waveform inside; this is the *jongly.aif* file. Notice also that the window has time markings on top. It appears that this file ends around *2800* milliseconds and there is blank space after the waveform because the buffer is *4000* milliseconds long. It would be great to get the buffer size to just about the same length as the audio file we're loading into it. We can obtain information about an audio file, such as how long it is, using the object *info~*.

10. Create a new object called *info~*
11. Open the Help file for *info~*

In the Help file, notice that the *buffer~* at the far right of the patch has only the name as its argument: *randall*. When an audio file is loaded into the *buffer~ randall*, the buffer will become the size of the audio file placed in it because it has not been given a buffer size in milliseconds.

12. Unlock the Help file patch and double click the *message read* connected to *buffer~ randall* change the text to *read jongly.aif*
13. Hold ⌘ (Mac) or ctrl (Windows) and click the *message read jongly.aif* to load this file into the *buffer~* object named *randall*

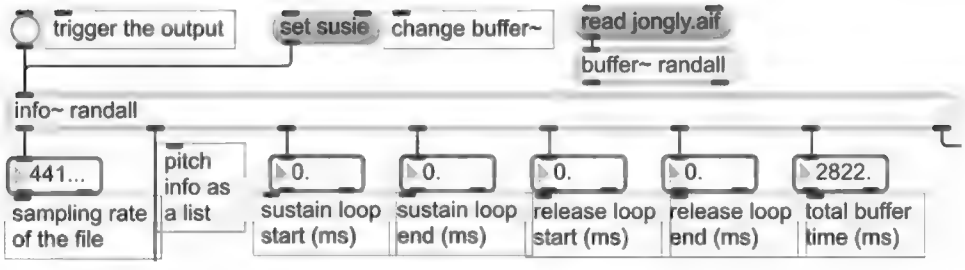


FIGURE 14.4

jongly.aif read into buffer~ randall to get info on audio file | info~ .maxhelp

Notice that the *info~* object also has the name *randall*. It is linked to the *buffer~* object with the same name. Thus, clicking on the *button* connected to *info~ randall* will get information about the audio inside of the *buffer~ randall* which contains *jongly.aif*.

14. Lock the patch and click the *button* connected to *info~* and notice that the *total buffer time* is displayed for the file *jongly.aif*: *2822* milliseconds and changes
15. Close the *info~* Help file without saving the changes

The *info~* object showed us that the length of the *jongly.aif* file is roughly *2822* milliseconds. Let's resize our *buffer~ mybuff* by sending it the *message size 2822*. We could also double click in the object and change the argument *4000* to *2822*, but instead, we'll explore one of the *message*s that *buffer~* understands.

16. Create a new message box called *size 2822* above the *buffer~* object
17. Connect the outlet of *size 2822* to the first inlet of *buffer~*
18. Click the *message* box *size 2822* to resize the buffer *mybuff* from *4000* milliseconds to *2822*
19. Hold ⌘ (Mac) or ctrl (Windows) and click the *message read jongly.aif* to reload the audio file into the resized buffer
20. Hold ⌘ (Mac) or ctrl (Windows) and double click the *buffer~* object to open the buffer window

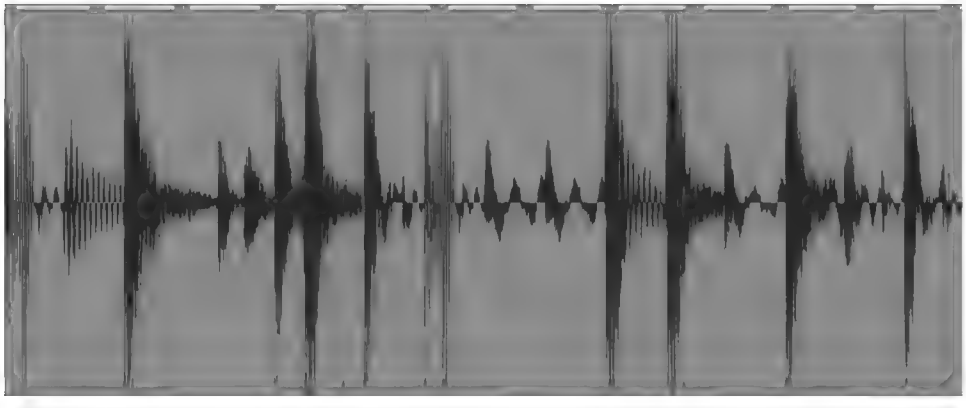


FIGURE 14.5
buffer window is resized to accommodate the length of the audio file | all_about_buffers.maxpat

Notice that the *jongly.aif* file now fits into the buffer. Now, let's hear what this *jongly* actually sounds like.

To play back audio from the buffer, you can use the *groove~* object with the same name argument as a *buffer~* object.

21. To the right of the *buffer~*, create an object called *groove~* with the argument *mybuff* (Be sure to leave some room around the *groove~* object)
22. Hold ⌘ (Mac) or ctrl (Windows) and double click the *groove~* object to reveal that this object refers to the same audio inside the *buffer~* object with the same name
23. Create a *gain~* object beneath *groove~*
24. Connect the first outlet of *groove~* to the first inlet of *gain~*
25. Create a new object called *ezdac~*
26. Connect the first outlet of *gain~* to both inlets of *ezdac~*
27. Create a new object called *meter~* to the right of *gain~*
28. Connect the first outlet of *gain~* to the inlet of *meter~*
29. Resize *meter~* so that it stands vertically like the *gain~* object

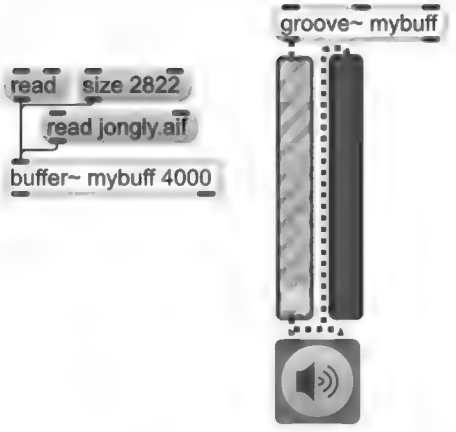


FIGURE 14.6

groove~ object named mybuff connected to output objects | all _about_buffers.maxpat

The *groove~* object is similar in some aspects to the way we used *sfplay~* and even understands some of the same *messages*, like *loop 1* to turn on looping playback.

30. Create a new *message* called *loop $1*
31. Connect the outlet of *loop $1* to the first inlet of *groove~*
32. Create a *toggle*
33. Connect the outlet of *toggle* to the first inlet of the *message loop $1*
34. Hold ⌘ (Mac) or ctrl (Windows) and click the *toggle* object to send the *message loop 1* to *groove~* enabling looping playback of the audio in the buffer

To play back the contents of *groove~* at normal speed, we need to send it a number *1*. It would stand to reason that we would just connect a *number* box to the first inlet of *groove~*. However, unlike *sfplay~*, *groove~* must receive the number as a signal instead of just a number. This allows the *groove~* to play back each sample of audio in the buffer and ultimately gives more control over the contents in the buffer. Converting a number to a signal is really easy: just connect a *flonum* box to a *sig~* object and you're done.

35. Create a new object called *sig~*
36. Connect the outlet of *sig~* to the first inlet of *groove~*
37. Create a *flonum* box
38. Connect the first outlet of *flonum* to the inlet of *sig~*

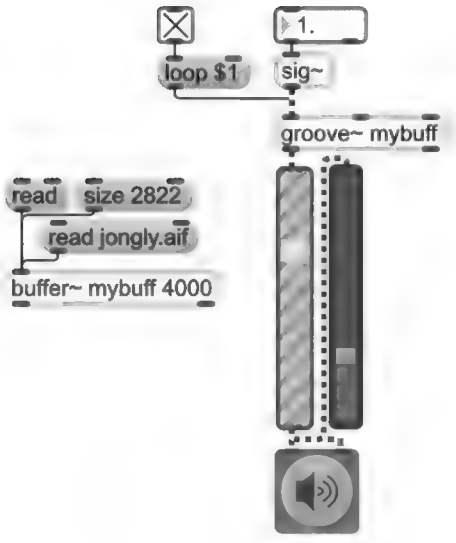


FIGURE 14.7

groove~ playing back at normal speed with loop enabled | all_about _buffers.maxpat

Entering a number into the floating-point number box will control the playback speed of the audio in the buffer through *groove~*. Entering a *1* in the floating-point number box will play the buffer at normal speed, while entering a *.5* will play it at half speed. Entering a *0* will not play back the file at all, while entering a *-.5* will play back the file in reverse at half speed.

39. Lock your patch and turn on the *ezdac~* object
40. Raise the *gain~* object half way
41. Enter the number *1* into the *flonum* box to hear *jongly.aif* play back at normal speed[1]
42. Slowly increase or decrease the value in the *flonum* box by whole numbers and decimals to change the playback speed of *jongly.aif*
43. Enter the number *0* to turn stop *groove~* from playing through the buffer

You have probably realized that you can make the audio sound "squeaky" by increasing the playback speed as well as decreasing the speed to achieve ominous sounds. We will discuss some other fun things to do with playback speed in a few moments. First, let's discuss how to record our own sounds into the buffer.

1. If you don't hear any playback and don't see any activity in *meter~*, reexamine the steps given and ensure that you followed the directions exactly. For example, make sure that looping was enabled. You may also try turning the *ezdac~* on and off.

Recording into a Buffer

Recording into a buffer is actually quite simple and similar to the way we used *sfrecord~* to record. First we need the components to get audio into our Max patch. Unlock your patch and

44. To the right of *groove~,* create a new object called *ezadc~*
45. Create a new object called *gain~*
46. Connect both outlets of *ezadc~* to the first inlet of *gain~*
47. Create a new object called *meter~* to the right of *gain~*
48. Connect the first outlet of *gain~* to the inlet of *meter~*
49. Resize *meter~* so that it stands vertically like the *gain~* object

Now that we have the necessary components to receive and monitor audio in our patch, we simply need to send the audio to a *record~* object with the same name argument as the buffer we want to record into.

50. Create a new object called *record~* with the argument *mybuff* as its name
51. Connect the first outlet of *gain~* to the first inlet of *record~ mybuff*
52. Create a *toggle*
53. Connect the outlet of *toggle* to the first inlet of *record~*

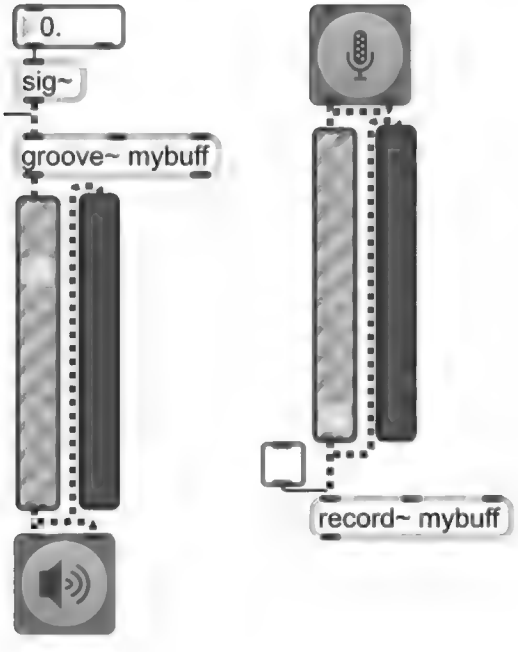


FIGURE 14.8
record~ named mybuff overwrites audio in the buffer when toggle is on | all_about_buffers.maxpat

The audio signal will start at *ezadc~*, flow through the *gain~*, and be sent to the *record~*. When we turn the *toggle* on, the audio received in *record~* will write over the space in the buffer. In fact, if you double click the *record mybuff* object, you will notice that it also brings up the buffer window. When we turn the *toggle* off, the recording in the buffer will cease. This will not in any way change the *jongly.aif* file, only the audio presently held in the buffer.

54. Lock your patch and turn on the *ezadc~* object
55. Raise the *gain~* slider so that the level at which you are speaking/singing shows up orange in the *meter~* object

56. Click the *toggle* on and begin speaking/singing to record into your buffer
57. Click the *toggle* off after about 3 seconds
58. Slowly increase or decrease the value in the *flonum* box by whole numbers and decimals to change the playback speed of what you just recorded into the buffer
59. Enter the number *0* to turn stop *groove~* from playing through the buffer

Remember that our buffer size is currently *2822* milliseconds. Anything recorded longer than this buffer size will wrap over to the beginning of the buffer. This can be a cool effect if you leave the recording *toggle* on while you speak/sing and manipulate the buffer in real time. You may also wish to increase the buffer size and the number of buffer~ objects in the patch to create a program that allows a user to record into multiple buffers and loop their playback.

If you are interested in continuing work with live looping and performance situations, you may want to consider "Max for Live," a collaborative software product by Cycling '74 and Ableton that allows Max to communicate directly with the DAW Ableton *Live*. If you want to use a DAW for live performance, using *Live* in combination with Max through "Max for Live" allows you to patch inside of *Live* which can yield very powerful results.

Max For Live

Max for Live (M4L) is an implementation of the Max/MSP/Jitter programming environment that is accessible directly from within the Ableton Live DAW, and allows you to write a Max program for MIDI, audio, or visualization for use inside of Live. M4L patches are routed within the MIDI or audio signal path within Live and appear alongside effects, instruments, and samples in Live's Detail view for each track. Writing patches for Live is done similarly to writing patches in Max. In M4L patches, MIDI objects (*midiin, midiout, notein, noteout*) are used to receive/send MIDI data from within Live, and the *plugin~* and *plugout~* objects are used to receive/send audio. M4L patches function just like Max patches with the advantage of being integrated into the Live DAW.

With Max for Live, several new objects (with the prefix *live.*) were added to the Max language and have special uses in M4L both visually and functionally. See the patch *M4L_equivalents.maxpat* in the *Chapter 14 Examples* folder for a list of M4L UI objects similar to Max objects we've discussed.

Referencing Playback Speed to a MIDI Note

60. With *groove~*'s playback stopped, click on the *toggle* connect to *record~* and record yourself singing a single long sustained note for about 3 seconds, then turn the *toggle* off
61. Check to make sure that your audio is able to play back from the buffer via *groove~* by entering the number *1* in the *flonum* box connected to *sig~*

At this point, the single note you recorded should be looping infinitely. Don't worry about critiquing your intonation as it plays back repeatedly; this is just a Max exercise. Suppose the note you were singing was the note middle C, MIDI note 60 (it's all right if it was not middle C; just pretend it was). It would be great to make a MIDI keyboard that plays your recorded note when middle C is played on your MIDI keyboard and transposes the same note up a whole step when the note D is played. It would be a synth/sampler made from your voice! Hang on for some math. Unlock your patch and

62. Create a new object called *notein*
63. Create a new object called *kslider*
64. Connect the first 2 outlets of *notein* to the 2 inlets of *kslider*
65. Create a *number* box beneath *kslider*'s first outlet
66. Connect the first outlet of *kslider* to the inlet of the *number* box
67. Create a new object beneath the *number* box called *mtof*
68. Connect the first outlet of the *number* box to the inlet of *mtof*
69. Create a *flonum* box beneath *mtof*
70. Connect the outlet of *mtof* to the inlet of the *flonum* box

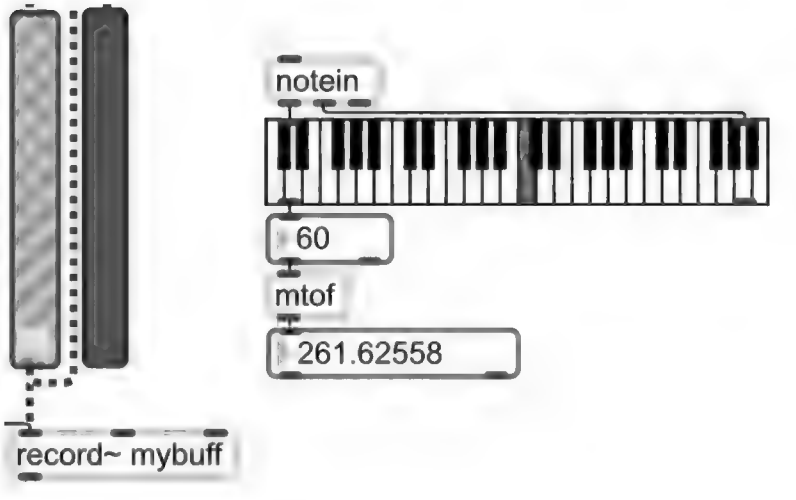


FIGURE 14.9
MIDI note 60 (middle C) displayed as frequency | all_about_buffers.maxpat

When you play middle C on your MIDI keyboard, the MIDI number *60* is converted to a frequency through *mtof* and displays the frequency number

261.62558 in the *flonum* box. Since the note we sang on our recording is middle C, we want our patch to play back our recorded sound at normal speed whenever we press the middle C key on our MIDI keyboard. Normal speed, as you know, is *1* sent to *sig~* and then *groove~*. So, if we divide the *mtof* frequency of each MIDI note by *261.62558*, when we play the MIDI note 60, we will essentially be dividing middle C's frequency, *261.62558* by the number *261.62558* which will give us the number *1;* any number divided by itself is equal to the number *1* (fifth grade math)

71. Create a new object called / with the argument *261.62558*
72. Connect the first outlet of the *flonum* box to the first inlet of / *261.62558*
73. Create a new *flonum* box beneath / *261.62558*
74. Connect the outlet of / *261.62558* to the inlet of the *flonum* box

The coolest thing about this math is that when you play the D above middle C, you will get the number *1.122462* which, if connected to the speed control of *groove~*, would transpose your note a whole step from the note you sang. In essence, if the note you originally sang was, in fact, middle C, you would be able to change the pitch of your note to whatever key you pressed on your MIDI keyboard.

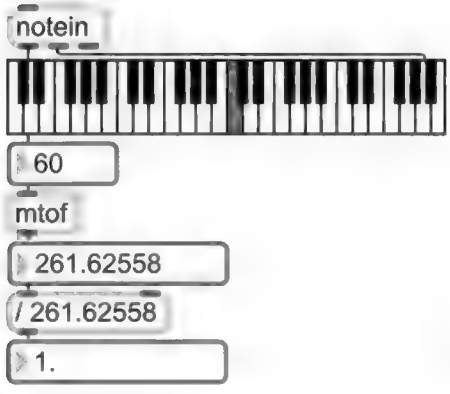


FIGURE 14.10
middle C frequency divided by itself yields 1 | all_about_buffers.maxpat

75. Connect the first outlet of the *flonum* box to the inlet of *sig~*

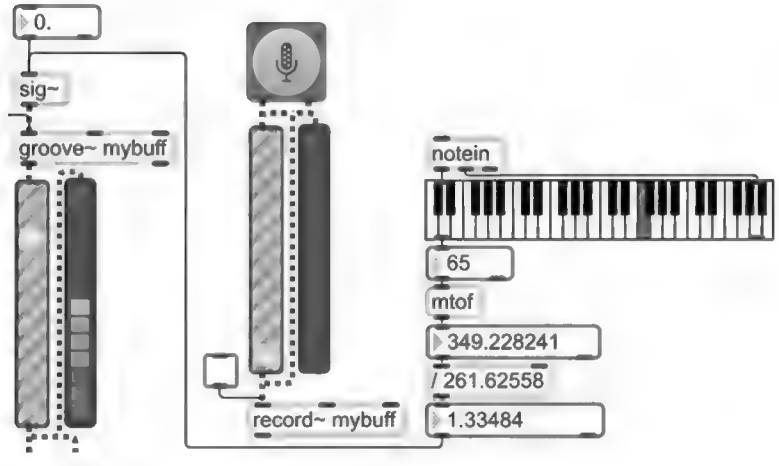


FIGURE 14.11
middle C plays contents of buffer at normal speed | all_about_buffers .maxpat

76. Lock your patch and play notes on your MIDI keyboard to change the playback speed of your recorded sound in correlation

to the MIDI pitch (try playing very high notes and very low notes as well to hear how this changes the sound)

77. Click *File>Save* and save this patch as *all_about_buffers.maxpat*

You can, in essence, make a synthesizer using your own voice as the timbre. Of course, the intonation may not be what you'd expect from a commercial software synthesizer, but it is a cool way to make a deeply personal instrument timbre from a single note. We can even make this timbre polyphonic. Let's take a look at an existing patch in the *Chapter 14 Examples* folder.

78. Open the file *polyphonic_buffers.maxpat* in the *Chapter 14 Examples* folder

Notice that the patch is similar to the *poly~* synth patch we made in Chapter 12. Nothing new here at all; just a different subpatcher inside of the *poly~* object in this patch. Unlock the patch and

79. Ctrl+click (Mac) or right click (Windows) on the *poly~* object and select *Object>Open Original "buffer_timbre"*

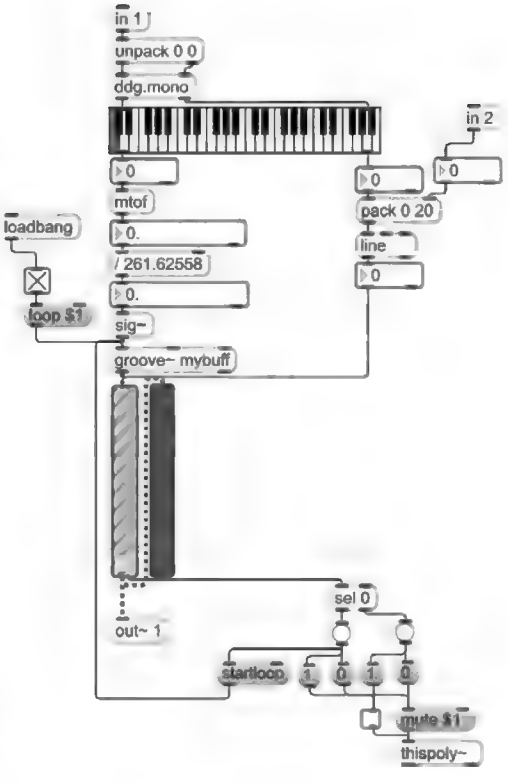


FIGURE 14.12
buffer used with poly~ | buffer_timbre.maxpat

Looking at the *buffer_timbre* patcher inside *poly~*, you can see that this patcher is really nothing new at all, either. It's constructed similarly to the *poly~* timbre patch we made in Chapter 12 but includes the MIDI to pitch and *groove~* playback portions of the patch we just made. The same *line* and *pack* components are used to control the velocity to gain functions, and the *thispoly~* muting business was copied directly from the timbre patch in Chapter 12. The only thing that is new is the inclusion of a *message* box containing the text *startloop,* which resets the starting playback point of the buffer every time a note is released. This ensures that when you play a chord on the MIDI keyboard, playback is never going to begin from the middle of any buffer; it will always start the loop from the beginning of the buffer. Close this window.

When the main patch loads, a small *.aif* file of me singing middle C called *vj_middle_C.aif* will be read into the buffer. You can then play polyphonic chords on your MIDI keyboard and hear the recording play back at different pitches. If you hold the chord long enough, you will hear the file loop back to the beginning. However, since the playback speed is different for each note of the chord, the return to the beginning of the file will be at different times and will resemble staggered breathing. Of course, you could always load up your own audio file of you singing a single continuous note for a longer period of time to avoid this phenomenon. The *record~* components of the patch have been included in the main patch as well.

Remember:

- Buffers are storage containers for audio.
- Using the filename *buffer~* allows other objects with the same name argument to work with the same audio inside the buffer.

New Objects Learned:

- *buffer~*
- *info~*
- *groove~*
- *sig~*
- *record~*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select MSP Tutorials from the Help menu and read
  - MSP Tutorial 14—Playback with Loops
  - MSP Tutorial 15—Variable-length Wavetable

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows a user to record into a buffer that is 5 seconds long. Create 3 different instances of the same buffer so that the recorded data can be played back polyphonically at different speeds. Use the mouse to control the playback speed and

volume of at least one of these 3 instances (hint: *mousestate* and *scale*). Allow the user to be able to record the entire performance of this patch as a sound file. In addition, you may use *line*, *random*, *counter*, and other objects to make changes in tempo and dynamics more interesting. You may also use more than one *buffer* if you'd like.

15

Audio Effects and Processing

In the last two chapters, we addressed ways to get live audio and sound files to play in your patch. In this chapter, we will address implementing audio effects into the patch.

Many DAWs (Digital Audio Workstations) allow you to add effects such as delays, reverb, and chorus into the audio signal path. Max, however, gives you complete control over the effects that you add since you build them yourself. For example, if you want to build a delay effect that delays the incoming audio signal by quarter notes for 5 seconds, sixteenth notes for 2 seconds, and eighth notes for 11 seconds, you may find that this very specific task is easier to carry out in Max than in the automation window of most DAWs. Similarly, if you wanted to perform some sort of filtering on audio in a buffer, you have complete control over the audio in your patch and the way that actions are carried out on it.

Since the nature of building effects is rather complex, we will focus on the concept of implementation using some simpler effects in the process. We will begin discussing *delays*, that is, combining an audio signal with a duplicate of the signal delayed by some amount of time. In addition to being a useful effect in itself, delays are the basis for other effects such as reverb, chorus, and flanging.

Preparing the Patch

Let's build a patch that will allow us to audition a few effects that we will build together. We will use some of the sound files that were installed on your computer when you installed Max (like *jongly.aif*) and *sfplay~* to play these files back. As you should know, you can just as easily implement these effects into a patch that uses live audio. Create a new patch and

1. Create a new object called *sfplay~*
2. Create a *toggle* above *sfplay~*
3. Connect the outlet of *toggle* to the first inlet of *sfplay~*
4. Create a *message* box containing the text *loop $1*
5. Connect the outlet of the *message loop $1* to the first inlet of *sfplay~*
6. Create a *toggle* above the *message loop $1*
7. Connect the outlet of *toggle* to the first inlet of the *message loop $1*
8. Create a *message* box containing the text *open jongly.aif*
9. Create a *message* box containing the text *open vibes-a1.aif*
10. Connect the outlet of both of these *message* boxes to the first inlet of *sfplay~*

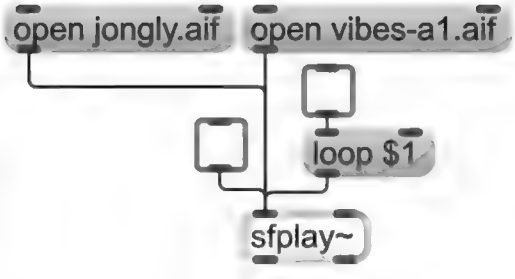


FIGURE 15.1
typical sfplay~ setup | effects_patch.maxpat

The patch now has the ability to play and loop either of the specified sound files; *jongly.aif*, as you know, is a drum loop, and *vibes-a1.aif* is a single vibraphone hit. Our next step is to send the signal of *sfplay~* to some audio output objects (*gain~*, *ezdac~*). However, we will use the *gate~* object to send our audio to different locations; we will be building different effects to receive the audio from each outlet of *gate~*.

11. Create a new object called *gate~* with the argument *4* to specify 4 outlets
12. Connect the first outlet of *sfplay~* to the second inlet of *gate~*

You probably remember the *gate* object from Chapter 7. The *gate~* object works similarly but handles audio instead of numbers. We have specified *4* outlets for *gate~* which means that we can have 4 different destinations for our audio file playback, and a different effect at each different destination. Like *gate*, send a number (in our case, 0–4) to choose the destination outlet for *gate~* to send the audio signal to.

13. Create a *number* box
14. Connect the first outlet of the *number* box to the first inlet of *gate~*

Let's use a *umenu* to simplify choosing a destination for *gate~*. We will be creating two different types of delay effects, and one equalization (EQ) example, so we can list the menu items for *umenu* as *off, direct output, delay 1, delay 2*, and *EQ*. Choosing the first *umenu* item *off* will output the number *0* which will shut off the *gate~* object. The *direct output* item will send a *1* to *gate~* allowing the *sfplay~* audio to travel directly from *gate~*'s first outlet to the output objects (*gain~*, *ezdac~*) with no effects (a "dry" signal). The remaining menu items will correspond to outlets 2–4 of *gate~* for which we will build two custom delays and one EQ.

15. Create a new object called *umenu*
16. Connect the first outlet of *umenu* to the inlet of the *number* box connected to *gate~*
17. Open the *Inspector* for the *umenu*
18. From the *Inspector* select *Edit* next to "Menu Items" and enter the following: off, direct output, delay 1, delay 2, EQ
19. Click OK and close the *Inspector*
20. Hold ⌘ (Mac) or ctrl (Windows) and click the *umenu* to choose a menu item and see a corresponding number sent to *gate~* to determine the output destination of the audio signal

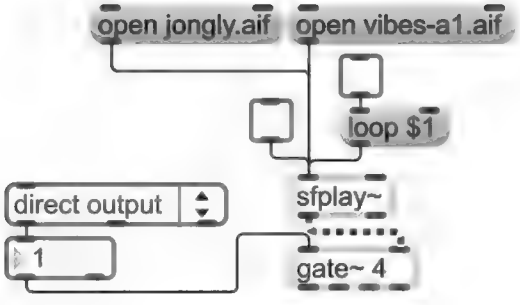


FIGURE 15.2
sfplay~ set to send signal to one of 4 different outlets | effects_patch .maxpat

21. Create a new object called *gain~*
22. Connect the first outlet of *gate~* to the first inlet of *gain~*
23. Create a new object called *ezdac~*
24. Connect the first outlet of *gain~* to the first two inlets of *ezdac~*

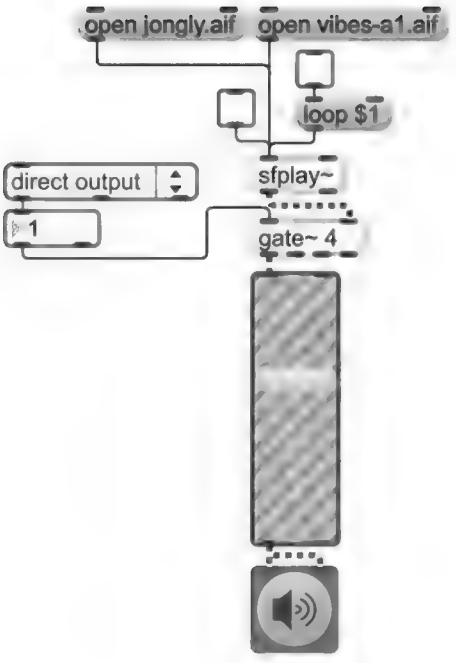


FIGURE 15.3

first sfplay~ outlet connects directly to gain~ and ezdac~ | effects_patch.maxpat

25. Lock your patch and click the *loop $1 toggle*
26. Click the *message jongly.aif*
27. Click the *toggle* connected directly to *sfplay~* so that it is on
28. Turn on the *ezdac~* and raise the *gain~* slider to a comfortable level (a little above half way)
29. Switch the *umenu* positions noting that only outlet 1 (labeled *direct output*) is connected to the *gain~* object
30. Turn off *ezdac~*

Delays

Now that our patch is able to play back audio, we can concentrate on creating some effects. The first effect we will create is a time-based delay in which a user can specify in milliseconds the amount of time to delay a signal. The objects we will use for this are *tapin~* and *tapout~*. Unlock your patch and

31. Create a new object called *tapin~* with the argument *3000*

The *3000* given as *tapin~*'s argument refers to *3000* milliseconds (3 seconds) of memory time to contain audio that is to be delayed. In a way, the *tapin~* object is like the *buffer~* object we discussed earlier in that it holds audio of some length. However, although 3 seconds of memory is given for *tapin~*, this doesn't mean that your audio input has to be under 3 seconds in order for the patch to work. The *tapin~* object simply contains 3 seconds of audio in its memory at one time, but it will delay any audio it receives as it comes through. The *tapout~* object is used to set the amount of delay time used to play back audio contained in the *tapin~* object.

32. Connect the second outlet of *gate~* to the inlet of *tapin~ 3000*
33. Create a new object called *tapout~* with the argument *150*
34. Connect the outlet of *tapin~ 3000* to the inlet of *tapout~ 150* (Note that this connection is not an audio patch cord. Instead, a data patch cord shows that the connection between these two objects is that they share the same audio in *tapin~*'s memory)
35. Connect the outlet of *tapout~ 150* to the first inlet of *gain~*

Our patch, as it is, will store an audio signal in the *tapin~* buffer and play it back *150* milliseconds later through *tapout~*. However, the real appeal of a delay line is combining the delayed signal with the original signal. For this, we will have to connect the original signal coming from *gate~* directly to the *gain~* object. Here, the dry signal will be combined with the delayed one.

36. Connect the second outlet of *gate~* to the first inlet of *gain~*
37. Create a *number* box
38. Connect the first outlet of the *number* box to the first inlet of *tapout~* (this allows us to change the number of milliseconds of delay time)

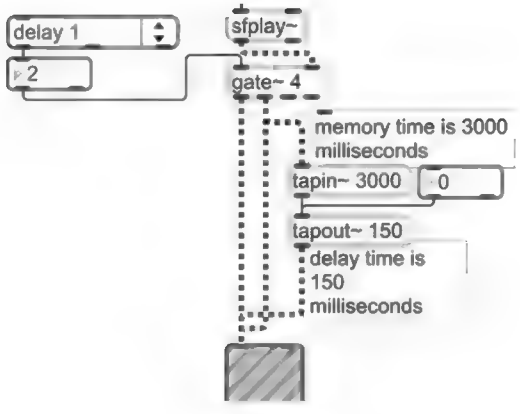


FIGURE 15.4
second sfplay~ outlet connects to gain~ and to tapin~/tapout~ delays | effects_patch.maxpat

39. Lock your patch and click the *message jongly.aif*
40. Click the *toggle* connected directly to *sfplay~* so that it is off and then on
41. Switch the *umenu* position to *delay 1*
42. Turn on *ezdac~*
43. Increase the numbers in the *number* box connected to *tapout~* to increase the delay time in milliseconds (set this number to *0* to hear no delay; just a doubling of the original signal)
44. Turn off the *ezdac~*

One of the appeals of using delays is that you can have multiple delay times. For example, if we made another *tapout~* with the argument *300*, we would have

the same signal repeating every *150* milliseconds and *300* milliseconds. Unlock your patch and

45. Create a new object called *tapout~* with the argument *300*
46. Connect the outlet of *tapin~ 3000* to the inlet of *tapout~ 300*
47. Connect the outlet of *tapout~ 300* to the first inlet of *gain~*
48. Create a *number* box
49. Connect the first outlet of the *number* box to the first inlet of *tapout~ 300*

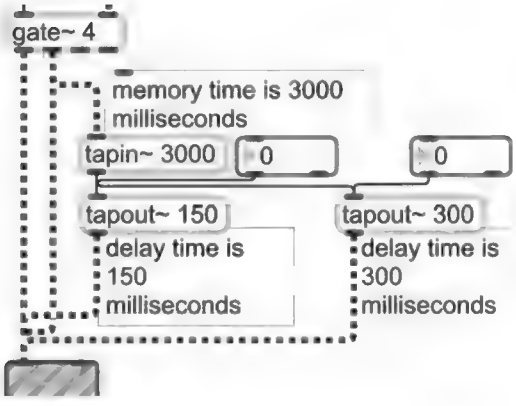


FIGURE 15.5
delays can be manipulated and multiplied | effects_patch.maxpat

50. Lock your patch and turn on *ezdac~*
51. Change the numbers in the *number* boxes connected to *tapout~* to *0* so there is no delay (just a tripling of the signal)
52. Change the numbers in the *number* boxes connected to *tapout~* to *150* and *300,* respectively
53. Click the *message vibes-a1.aif*
54. Click the *toggle* connected directly to *sfplay~* so that it is off and then on
55. Switch the *umenu* position to *direct output* to observe the dry sound of *vibes-a1.aif*
56. Switch the *umenu* position to *delay 1* to introduce delays to the sound of *vibes-a1.aif*
57. Increase and decrease the numbers in the *number* boxes connected to *tapout~*
58. Turn off the *ezdac~*

As you can see, adding more *tapout~* objects will give you more delay lines. However, you can also add additional arguments to a *tapout~* object to set more delay lines. Doing so will also add signal outlets for *tapout~*. In Figure 15.6, notice that the first *tapout~* object has the arguments *150* and *300* and, as a result, has two outlets: the first sends the signal after a *150* millisecond delay, and the second sends the signal after a *300* millisecond delay. Note that the

delay time for *tapout~* cannot exceed the memory time given as an argument for *tapin~*.

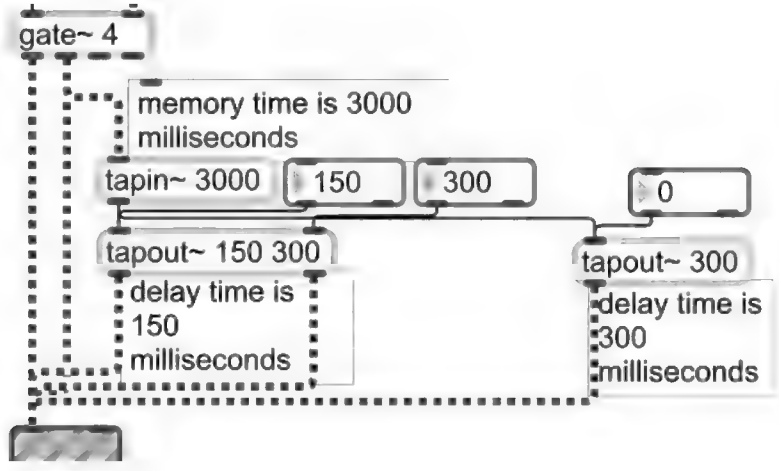


FIGURE 15.6
more delays | effects _patch.maxpat

Another Way to Delay

The example we just looked at was a time-based delay. However, we can use the *delay~* object to delay a signal based on samples or tempo-related time intervals. In the latter case, a delay line can be set to, for example, quarter notes at a certain tempo specified in the *transport*.

In the former, the term *samples* does not refer to sampling in the sense that a MIDI note may trigger back some "sampled" (prerecorded) audio file (like using the woodwind samples in your favorite notation program). Instead it refers to a rate by which an audio signal—for example, you singing the note A, which is a continuous flow of audio—can be broken down into "discrete" and evenly spaced signals. In digital audio, each of the discrete signals is given a numerical value and these represent the overall continuous signal that was broken down. Each of these values is referred to as a sample.

There are different numerical rates by which someone can sample audio. For example, many computer sound cards can record 44,100 samples per second. This means that for a one-second recording of, for example, once again, you singing the note A, there are 44,100 samples representing that one second of continuous audio. Some more advanced sound cards have the ability to use more samples per second in the sampling process.

The *sampling rate* is different from a *bit rate* which refers to the amount of binary digits or *bits*, for short, that can be used to represent the data being recorded. For example, as you may know, "binary code" refers to a counting system using only the numbers *1* and *0* where *1* represents that something is in the state of being "on" and *0* represents that something is "off." The more *1*s and *0*s you string together, the longer the number will be and the more information your number will contain—information that is used to describe something, such as amplitude, in computer terms. For example, the binary "word"

1011101011110001 represents many more "on/off" states than the word *1011*. The longer binary word can better represent something than the shorter one simply because it has more numbers (binary digits or *bits*) to work with.

When a binary word is 16 characters long, it is said to be *16-bit*; 32 characters is *32-bit*, and so on. A *1-bit* word has only one character and two possible states: *1* or *0*. A *2-bit* word has two characters and 4 possible combinations of *1s* and *0s*. An *8-bit* word has 8 characters and can represent *256* possible values given the combination of *1s* and *0s*. The reason that MIDI uses the numbers *0–127* is because the protocol dedicates 7 bits to representing things like pitch and velocity; 7 bits yields *128* possible combinations of *1s* and *0s*.

All of this to say that recording something at *24-bit* means that you are using more numbers to digitally represent the recorded analog signal when it is converted from analog to digital than if you recorded it at *16-bit*. The sampling rate refers to how frequently you will take "snapshots" (samples) of a continuous audio signal. Whew!

The *delay~* object takes a maximum delay value in samples as its first argument and an actual delay value in samples as its second argument. The delay is then based on samples received instead of time in milliseconds as we used in *tapin~* and *tapout~*.

59. Create a new object called *delay~* with the arguments *44100* and *500*
60. Connect the third outlet of *gate~* to the first inlet of *delay~*
61. Connect the outlet of *delay~* to the first inlet of *gain~*

Since we want to reflect a delay of the signal coming from *gate~*'s third outlet, we should also connect the third outlet directly to *gain~*.

62. Connect the third outlet of *gate~* to the first inlet of *gain~*

In this example, the internal memory of the *delay~* object will be *44100* samples. If my computer's sound card is set to sample at a rate of *44,100* samples per second, the internal memory can be thought of as being one second, though it is actually measured in samples and not time. The audio signal will be delayed *500* samples as indicated by the second variable.

If at this point you're saying "Enough of this complicated sample rate bologna; I'll just stick with *tapin~* and *tapout~*," let me try to sell you on the fact that since *delay~* is not stuck dealing strictly with milliseconds and time, it can also be used in conjunction with rhythmic values from the *transport*.

63. Create 2 *message* boxes containing the text *0* and *4n*, respectively
64. Connect the outlet of each of these *message* boxes to the second inlet of *delay~ 44100 500*

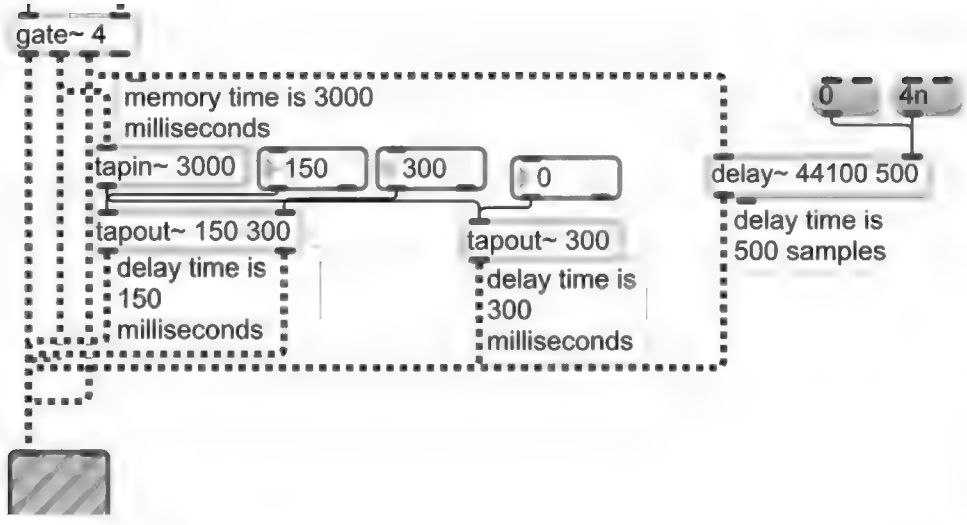


FIGURE 15.7

third sfplay~ outlet connects to gain~ and to delay~ object | effects _patch.maxpat

65. Create a new object called *transport*
66. Lock your patch and turn on *ezdac~*
67. Click the *message vibes-a1.aif*
68. Switch the *umenu* position to *delay 2*
69. Notice that the sound is delayed *500* samples
70. Click the *message 4n* to change *delay~*'s second argument to quarter notes at the tempo specified in the *Global Transport*
71. Double click the *transport* object
72. Change the tempo in the *Transport window* and observe that the delay times change
73. Turn off the *ezdac~*

As you can see, this type of tempo-relative delay can be advantageous over *tapin~* and *tapout~* in certain contexts. Now you know two different ways to make delays. Notice that both ways of implementing these effects are basically the same: insert the effect at some point during the "signal chain" before the final output stage (*gain~* and *ezdac~*). There are numerous effects and signal processing objects and abstractions which, though they process the sound differently, can all be implemented in pretty much the same way. In addition, if you know a bit of signal theory, you can use delays to create other effects.

EQ

One object that we definitely need to discuss is the *cascade~* object when used in conjunction with *filtergraph~*. These two objects can be used to provide a graphic equalizer (EQ) in your patch. An EQ can be used to raise or lower the volume of some groups of frequencies known as "bands" similarly to the way that you can increase or decrease the bass and treble in your car stereo.

74. Create a new object called *cascade~*
75. Connect the fourth outlet of *gate~* to the first inlet of *cascade~*
76. Connect the outlet of *cascade~* to the first inlet of *gain~*
77. Create a new object called *filtergraph~*
78. Connect the first outlet of *filtergraph~* to the second inlet of *cascade~*
79. Open the Help file for *filtergraph~*
80. Unlock the *filtergraph~* Help file and copy the *umenu*
81. Paste the *umenu* from the Help file into your patch and close the Help file
82. Connect the second outlet of *umenu* (the outlet that outputs the menu item text) to the first inlet of *filtergraph~*

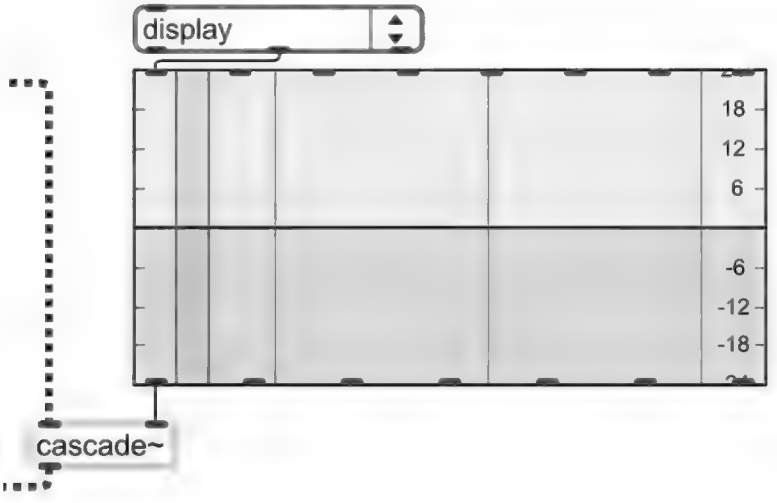


FIGURE 15.8
fourth sfplay~ outlet connects to cascade~ | effects_patch.maxpat

83. Lock your patch and click the *message jongly.aif*
84. Click the *toggle* connected directly to *sfplay~* so that it is off and then on
85. Switch the *umenu* position to *EQ*
86. Turn on *ezdac~*
87. While *jongly.aif* is playing, choose one of the EQ types from the *umenu* connected to *filtergraph~*
88. Click in the center of the *filtergraph~* object to move the center red rectangle (the bandwidth) horizontally. Clicking above the edge of the rectangle will allow you to resize the red rectangle
89. Turn off the *ezdac~*

In this patch, *filtergraph~* is like the graphical control for using the *cascade~* object. These objects can be particularly useful if you want to tweak the timbre of some recorded audio as well as being useful for conducting research in hearing and perception. Again, several frequencies that are grouped together are known as a "band," and thus the term "bandwidth" refers to the size encom-

passing some number of frequencies within a band. A *bandpass* filter allows only frequencies within the band to pass through *cascade~* whereas a *highpass* filter allows only high frequencies to pass, and so on. You can resize the shape of the *filtergraph~* to control the way certain frequencies will be filtered from the original sound source.

EQ isn't really an effect as much as it is a volume enhancement or reduction of some frequencies in the signal that are already there. In fact, we can send "white noise," a sound in which every audible frequency is present at an equal volume, to *cascade~* and use *filtergraph~* to filter out the frequencies we don't want in order to create new timbres. To create white noise, we'll use the *noise~* object. Unlock your patch and

90. Create a new object called *noise~*
91. Create a new object called *gain~*
92. Connect the outlet of *noise~* to the first inlet of the newly created *gain~* object
93. Connect the outlet of *gain~* to the first inlet of *cascade~*

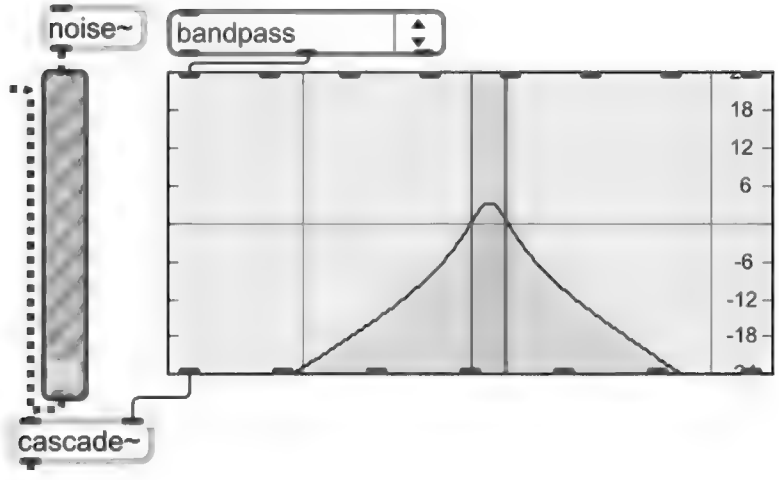


FIGURE 15.9

white noise is sent from noise~ to cascade~ and filtered with filtergraph~ | effects_patch.maxpat

94. Lock your patch and uncheck the *toggle* connected to *sfplay~* so that *jongly.aif* is not playing
95. Turn on the *ezdac~*
96. Slowly raise the *gain~* receiving a signal from *noise~* to allow white noise to enter *cascade~*
97. Make changes to *filtergraph~* effectively creating new timbres from noise
98. Turn off the *ezdac~*

This type of filtering is known as "subtractive synthesis" since all of the frequencies are present in white noise and we're actually removing frequencies from the sound. It's kind of like sculpting with sound. Think about how interesting it

could be to compose a piece of music using only white noise. One way to compose such a piece might be with the *preset* object.

As the name suggests, the *preset* object is used for storing preset states of UI objects in a patch. While not as robust with features as the *pattr*-style presets we stored earlier, *preset* is a nice quick way to store different values in a nice-looking graphical object. Let's use *preset* to store a few different *filtergraph~* settings.

99. Unlock your patch and create a new object called *preset*
100. Connect the first outlet of *preset* to the first inlet of *filtergraph~*

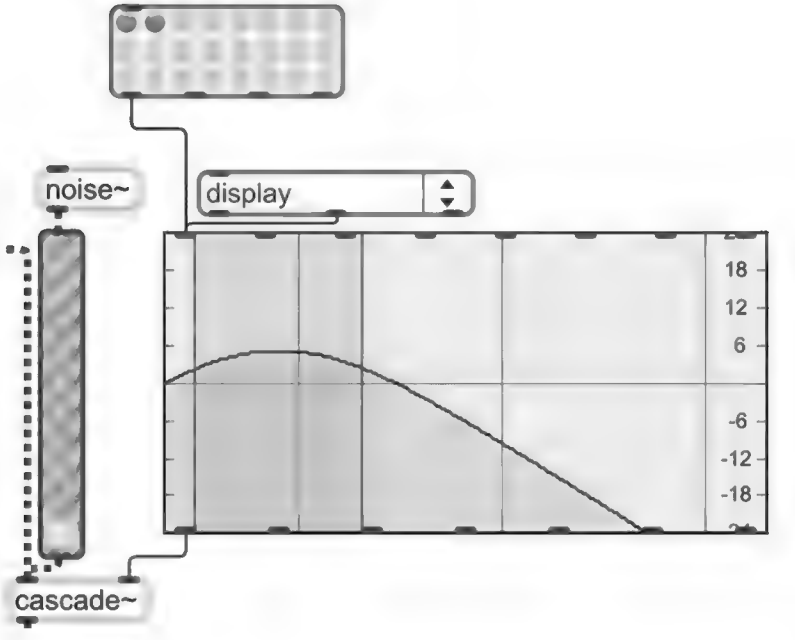


FIGURE 15.10

preset object is used to recall different settings for filtergraph~ | effects _patch.maxpat

To store the state of any UI object, connect the first outlet of *preset* to the first inlet of the object. When the UI object is connected, its state can be stored in *preset* by holding the *shift* key and clicking in one of *preset*'s bubbles while the patch is locked.

101. Lock your patch, hold the *shift* key down while clicking in the top right bubble of the *preset* object to store the current setting of *filtergraph~* into the *preset* object
102. Change the bandwidth or curve type of *filtergraph~* in some way and, once again, hold the *shift* key down while clicking in a different *preset* bubble
103. Click both bubbles to see the state of *filtergraph~* alternate between both presets you made

Interestingly enough, the *preset* object, if not connected to any objects, can be used to save the state of all UI objects in a patch. Use caution with this fea-

ture, however, as it can be easy to forget about "that one *umenu* or *toggle*" that you'd rather not have included in the preset.[1]

A number, starting at *1*, sent to *preset*'s inlet can be used to quickly recall a preset. You may want to try saving a bunch of different *preset* states and then using an object like *counter* to move through each preset.[2] Click *File>Save* and save this patch as *effects_patch.maxpat*. Close the patch.

Other Effects

There are many ways to create and work with audio effects in Max. If you already have sound processing software that you like, you may want to consider using objects like *vst~* which allows you to load your favorite VST plug-in into a Max patch, and *rewire~* which allows you to transmit audio data to other audio programs using the Rewire audio protocol. As mentioned, "Max for Live" allows you to use *Live* to host all of your favorite sound engines with Max as the front-end.

Another option for sending sound from Max to other applications is the use of third party software like *Soundflower,*[3] a free audio routing utility for Mac computers that allows you to send audio from Max to other audio programs similarly to the way that we sent MIDI "from Max" in Chapter 11. Another popular and free utility is *Wormhole2*[4] for both Mac and Windows platforms.

You may find it helpful, at this point, to explore the example patches bundled in the *examples\effects* folder in the Max application folder. These patches contain many great examples of working effects like reverb, vocoding, and comb filtering, which you can adapt into new or existing patches.

Remember:

- Recording something at *24-bit* means that you are using more numbers to digitally represent the recorded analog signal when it is converted from analog to digital than if you recorded it at *16-bit*.
- The sampling rate refers to how frequently you will take samples of a continuous audio signal.

New Objects Learned:

- *gate~*
- *tapin~*

1. Connecting *preset*'s third outlet to a UI object excludes the object's state from being stored in a preset.
2. See the file *effects_patch.maxpat* in the *Chapter 15 Examples* folder for a demonstration of using *counter.*
3. Soundflower: http://code.google.com/p/soundflower/.
4. Wormhole2: http://code.google.com/p/wormhole2/.

- *tapout~*
- *delay~*
- *cascade~*
- *filtergraph~*
- *noise~*
- *vst~*
- *rewire~*
- *preset*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select MSP Tutorials from the Help menu and read
  - MSP Tutorial 27—Delay Lines
  - MSP Tutorial 28—Delay Lines with Feedback
  - MSP Tutorial 29—Flanging
  - MSP Tutorial 30—Chorus
  - MSP Tutorial 31—Comb Filter

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows a user to record into a buffer that is 5 seconds long and uses some sort of control interface (mouse, keyboard, *transport*, time) to process the sound in novel (but planned) ways. Allow the user to be able to record the entire performance of this patch as a sound file. You may also use more than one *buffer* if you'd like.

16

Working with Live Video

In this chapter, we will discuss *Jitter,* a set of objects that handles video and visual-related content in Max. Jitter is for handling video what MSP is for handling audio. Adding visual elements to your patch can turn your interactive systems into multimedia ones. By the end of this chapter, you will have created a patch that changes the color of a live video when pitches are played.

In this book, we will not cover all the features and options available with Jitter. Instead, we will focus on ways to incorporate video from cameras and existing video footage into our patches. This will address some of the immediate connections between music education and video while providing a broad enough foundation of Jitter concepts and techniques to build on later through the Jitter tutorials and exploration.

Matrix

In the same way that the MSP objects end with ~, the Jitter objects all begin with "*jit.*" In fact, you already learned one Jitter object in Chapter 9, *jit.cellblock*; the object that resembled a spreadsheet. A spreadsheet is a type of grid known as a *matrix* in which each location on the grid, like a spreadsheet cell, contains some (or no) information. When we discussed the *mousestate* object and the amount of pixels determined by the screen resolution, we were discussing a matrix—a matrix (a grid) that displays a certain number of pixels (like tiny little cells in a spreadsheet that display color).

The more pixels that are used to represent an image, the higher the resolution of the image will be, resulting in a clearer appearance. If, for example, you had a photo of the *Mona Lisa* that only used 12 pixels to represent the immense detail of the painting, your photo would appear less than lifelike, to say the least. Similar to our discussion last chapter about *bit depth*, the higher the resolution you can work with, the more clarity your image will have, generally. Of course, the trade-off is that there is a greater demand on your computer's resources in terms of memory, processor speed, video processing (graphics card), and hard drive space (for storing these large videos).



FIGURE 16.1

high resolution picture (left) compared to a low resolution version (right) in which the number of pixels can actually be counted

Camera Input

Let's get some data from our computer's camera.[1] We will accomplish this with an object called *jit.qt.grab*. I've already commented on the *jit* prefix for this object; the *qt* part refers to Apple *Quicktime* components for visual playback which Jitter relies upon for some of its objects. As a result, you should have Quicktime installed on your machine (Mac or Windows) prior to using Jitter objects.[2]

On Windows, however, some things are a bit different at times. For example, Windows does not have Quicktime built into the operating system as Macs do, so instead, we'll use the object called *jit.dx.grab* where the *dx* refers to Microsoft *Direct X* components for visual playback. Most of Jitter's objects work without inconsistencies between platforms; this object is one exception.[3] We will use the *jit.qt.grab* object in the illustration below, but *jit.dx.grab* should be used instead by Windows users.

1. If your computer doesn't have a camera built in, you may want to purchase a webcam in order to complete this unit.
2. Quicktime is free and available from http://www.quicktime.com.
3. However, *jit.qt.grab* will run on Windows with a third party "vdig." See Help file for more details.

1. Create a new patch
2. Create a new object called *jit.qt.grab* or *jit.dx.grab* if you are using Windows (*jit.qt.grab* will be shown in the illustrations to follow)
3. Create a new *message* box containing the text *open*
4. Connect the outlet of *message open* to the first inlet of *jit.qt.grab*

When the *open* message is received by *jit.qt.grab*, the camera(s) connected to your computer will be initialized.

5. Hold ⌘ (Mac) or ctrl (Windows) and click the *message open* to initialize your camera

Even though *jit.qt.grab* is ready to start getting a video signal from your camera, we're still missing a few major components in this patch. One of them is a window of some sort to display the image taken from the camera. This object is the *jit.pwindow* object.

6. Create a new object called *jit.pwindow* beneath *jit.qt.grab*
7. Connect the first outlet of *jit.qt.grab* to the inlet of *jit.pwindow*



FIGURE 16.2
jit.qt.grab object connected to jit.pwindow | color_changer.maxpat

You can have only one *jit.qt.grab* object running at one time per camera. This doesn't mean, however, that you can't receive a single video signal into multiple windows or use multiple cameras simultaneously.

Notice that the patch cord between two Jitter objects looks different from a Max patch cord and an MSP patch cord. The thick green cord shows that there is some sort of video matrix information being sent. The *jit.qt.grab* object will "grab" sequential video images from your camera and display them in *jit.pwindow*.[4] However, the *jit.qt.grab* object needs to be told how often to "grab" images from your camera. This rate is referred to as a frame rate.

If you were to send a bang from a *button* to the *jit.qt.grab* object, a single snapshot image would be taken from your camera and displayed in the *jit.pwindow*. You may be saying to yourself, a *metro* object can be used to send bangs on a more scheduled basis. Instead of using *metro*, let's use a similar object

4. Note that *jit.window* is an object that works like *jit.pwindow*, but displays the image in a floating window as opposed to a "patcher window."

called *qmetro* that runs in low priority in the background, thus allowing more freedom for other processes like digitizing video.

8. Create a new object called *qmetro* with the argument *30*
9. Connect the outlet of *qmetro 30* to the inlet of *jit.qt.grab*
10. Connect the first outlet of *jit.qt.grab* to the inlet of *jit.pwindow*
11. Create a *toggle*
12. Connect the outlet of *toggle* to the first inlet of *qmetro 30*

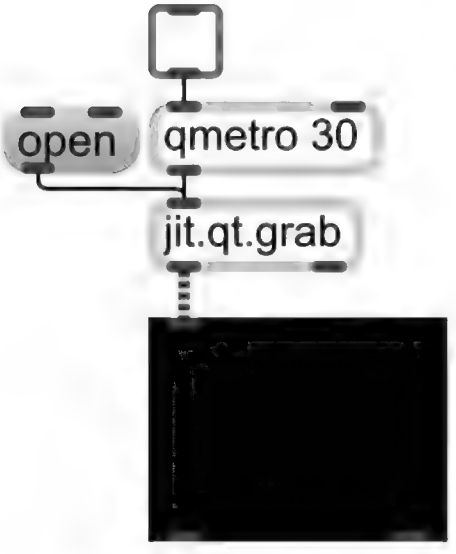


FIGURE 16.3
basic objects used get video from a camera into Max| color_changer .maxpat

13. Lock your patch and click the *toggle* to turn it on, which will begin displaying video from your camera[5]

A bang will be sent to *jit.qt.grab* every *30* milliseconds displaying a new image in *jit.pwindow*. You can resize the *jit.pwindow* by clicking and dragging in the lower right-hand corner of the object. Unlock your patch and

14. Resize the *jit.pwindow*

Adjusting Color

I suppose one of the easiest things to do with video input is to adjust the color. The camera will try its best to reproduce the actual colors it is aimed at, but we can tweak those colors a bit (or a lot) by using the *jit.scalebias* object. With this object, you can "bias" the color toward some amount of red, green, or blue, or what are known as RGB (red, green, blue) values. We will want to insert this object between the *jit.qt.grab* (receiving the image from the camera) and the *jit*

5. If you do not see video in the *jit.pwindow*, ensure that you have repeated each step including clicking the *open* message to *jit.qt.grab*. Then, consult the Max window to see if there are any errors. You may need to restart Max if you plugged your camera into the computer after Max was launched.

.pwindow (displaying the image from the camera) so that our "biased" color will be displayed in the window.

15. Create a new object called *jit.scalebias*
16. Disconnect the patch cord from the first outlet of *jit.qt.grab* and the inlet of *jit.pwindow*
17. Connect the first outlet of *jit.qt.grab* to the inlet of *jit.scalebias*
18. Connect the outlet of *jit.scalebias* to the inlet of *jit.pwindow*



FIGURE 16.4

jit.scalebias used to bias the color of the image before displaying it in jit .pwindow | color_changer .maxpat

By sending *jit.scalebias* a *message* box containing the text *rbias 1*, the color of the image from the camera will be biased toward the color red (the *r* in *rbias* is for red). We can also send *messages* for *gbias* (green) and *bbias* (blue) to add further color biasing.

19. Create 3 *message* boxes containing the text *rbias $1, bbias $1,* and *gbias $1,* respectively
20. Connect the outlet of each *message* box to the inlet of *jit. scalebias*
21. Create 3 *flonum* boxes above the *message* boxes
22. Connect the first outlet of each *flonum* object to the inlet of each *message* box, respectively

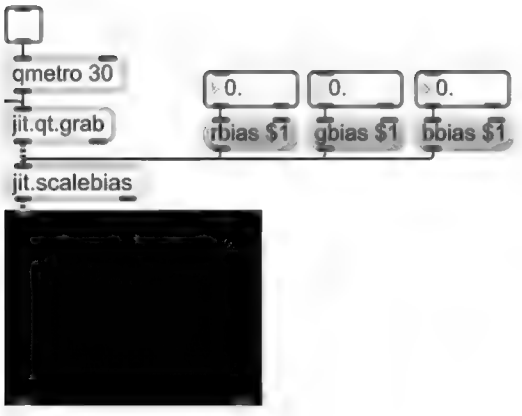


FIGURE 16.5

messages sent to jit.scalebias to adjust RGB values | color_changer.maxpat

23. Lock your patch and increase the numbers in the *flonum* boxes between *0.* and *1.* to see the image color change

As you changed the number values to yield different colors, you may have noticed that the change in colors is not very gradual. We can, as you know, fix this by using the *line* object to change values at a set speed. Unlock your patch and

24. Create a new object called *line* with the argument *0.* (don't forget the period)
25. Connect the first outlet of *line 0.* to the inlet of the *flonum* connected to the *rbias $1 message* box
26. Create a new object called *pack* with the arguments *0.* and *1000* above the *line* object (again don't forget the period)
27. Connect the outlet of *pack 0. 1000* to the inlet of *line*
28. Create a *flonum* box above *pack*
29. Connect the outlet of the *flonum* box to the first inlet of *pack*
30. Highlight all of the objects created in steps 23–28 (*line, pack, flonum*) and copy/paste them above the *gbias $1 message* box and again above the *bbias $1 message* box
31. Connect the first outlet of the 2 newly created *line* objects to the inlet of the *flonum* box above the *gbias* and *bbias message* boxes

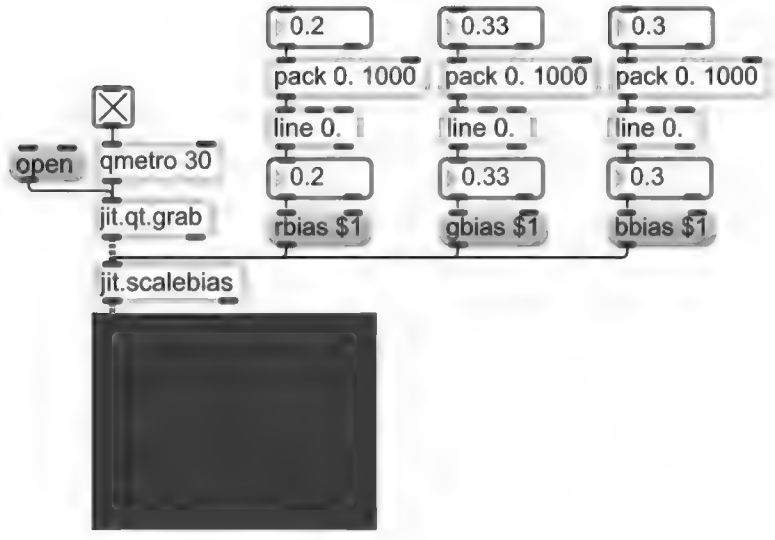


FIGURE 16.6

line used to smooth the transition from one color to the next | color _changer.maxpat

Now, when a value is entered into the floating-point number boxes above the *pack* objects, *line* will ramp to the desired value over the course of *1000* milliseconds (because of *pack*'s second argument) causing the bias of RGB values to gradually increase and decrease (because of the *jit.scalebias*). This will make the color change much more gradual than before. You can feel free to increase or decrease the value in *pack*'s second argument to any number you like.

32. Lock your patch and change the values in the *flonum* boxes above *pack* to see the color of the camera image change gradually over 1 second

You have probably now realized that changing 3 different number boxes and guessing what the color will be is kind of annoying. There is an object called *swatch* that can make selecting a desired color much easier. Unlock your patch and

33. Create a new object called *swatch*

The *swatch* object allows a user to click on a graphical display of colors and receive RGBA values as a list from its outlet. The values will be between *0.* and *1.* with the *Inspector* option to view the values according to the common *0–255* color scale. We've already discussed the letters R, G, and B, but not the A, or *alpha* channel. Alpha, in most cases, is used to describe opacity. If the alpha value for a pixel is at *0*, the pixel is transparent. In many cases, the A value in RGBA will remain at *1* when using *swatch*. Since *swatch* outputs a list of 4 floating-point numbers, we will have to use *unpack* to get each RGB value to the corresponding *bias* message. We should also create a new bias *message* called *abias* to accommodate the alpha channel.

34. Create a new object called *unpack* beneath *swatch* with the arguments *0. 0. 0. 0.* (don't forget the decimal point after each *0*)

35. Connect the first outlet of *swatch* to the inlet of *unpack 0. 0. 0. 0.* beneath it
36. Connect the first outlet of *unpack* to the inlet of the *flonum* box indirectly connected to the *rbias $1* message
37. Connect the second outlet of *unpack* to the inlet of the *flonum* box indirectly connected to the *gbias $1* message
38. Connect the third outlet of *unpack* to the inlet of the *flonum* box indirectly connected to the *bbias $1* message
39. Copy the *bbias $1 message* box and all of the objects connected to it (*flonum, line, pack, flonum*) and paste a copy of this entity to the right of the existing *bbias* message
40. Connect the last outlet of *unpack 0. 0. 0. 0.* to the inlet of the newly copied *flonum* box
41. In the newly copied *message* box, change the text of the *bbias $1* to *abias $1* in order to send alpha bias messages to *jit.scalebias*
42. Connect the outlet of the *message abias $1* to the inlet of *jit. scalebias*

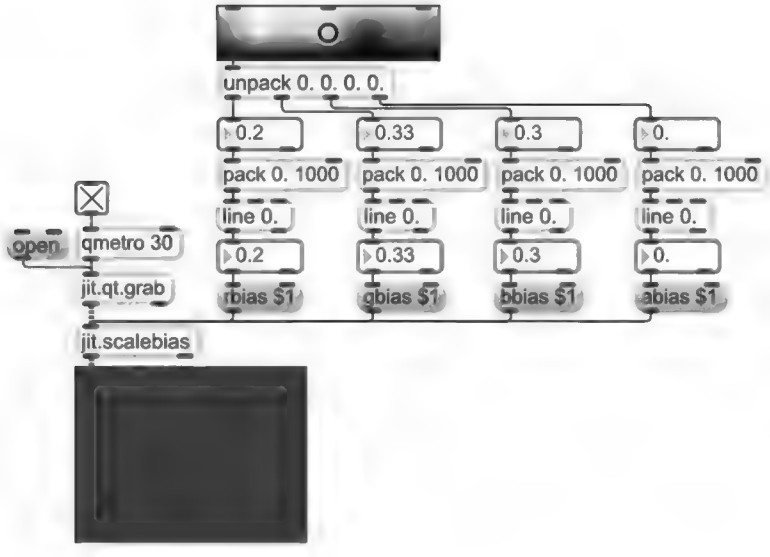


FIGURE 16.7

swatch allows the user to select a color by which the image will be biased | color_changer.maxpat

43. Lock your patch and click on different locations in the *swatch* to see the *jit.pwindow* color reflect the *swatch* color

Mapping MIDI Pitches to Color

Some individuals—Olivier Messiaen, for example—claim to see colors when they hear music, a condition known as *synesthesia*. Whether your brain does this or not, it can be a cool idea to map certain colors to pitches, especially when the color that's changing is a video image of yourself playing the keyboard. Let's

allow this patch to bias the video image toward a particular color whenever a certain note is played on our MIDI keyboard.

To begin, let's establish 12 colors that we find pleasing and store them with a *preset* object. Unlock your patch and

44. Create a new object called *preset*
45. Connect the first outlet of *preset* to the inlet of *swatch*
46. Lock your patch, and move the *swatch* to a color that you find pleasing, then hold the *shift* key and click in the first *preset* bubble
47. Repeat the previous step until you have filled the first 12 bubbles of *preset* with different colors that you like
48. Create a *number* box above *preset*
49. Connect the first outlet of the *number* box to the inlet of *preset*

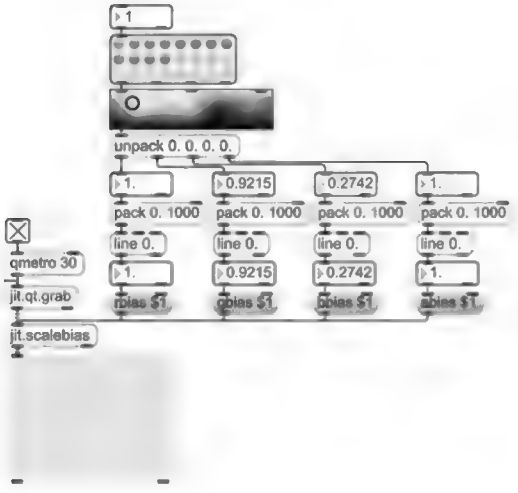


FIGURE 16.8

preset is used to store 12 swatch settings | color_changer.maxpat

The next few steps might be obvious to you: we'll need to get some notes in from our MIDI keyboard and map a certain pitch class to trigger a preset. Can you conceptualize a way to do this?

50. Create a new object called *notein*
51. Create a new object called *kslider*
52. Connect the first 2 outlets of *notein* to the 2 inlets of *kslider,* respectively
53. Create a *number* box beneath the outlets of *kslider*
54. Connect each outlet of *kslider* to the inlet of the *number* box beneath it
55. Create a new object called *noteout*
56. Connect the first outlet of the first *number* box to the first inlet of *noteout*
57. Connect the first outlet of the second *number* box to the second inlet of *noteout*

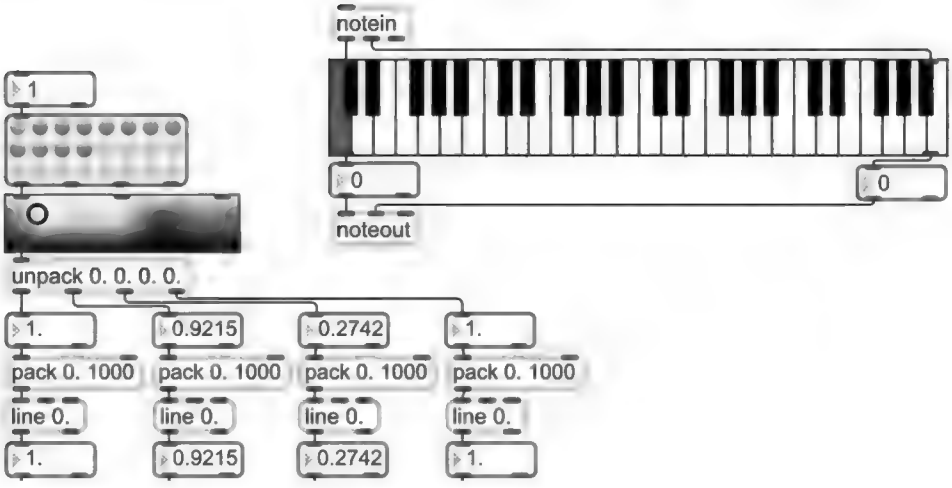


FIGURE 16.9

basic MIDI I/O objects added | color_changer .maxpat

You've just made a MIDI synth; nothing new so far. The next step is to reduce all incoming pitches to pitch class values 0–11 so that any C, regardless of octave, will trigger the same preset color. Do you remember how to reduce pitches to a pitch class?

58. Create a new object called % with the argument *12*
59. Connect the first outlet of the *number* box receiving from *kslider*'s first inlet to the first inlet of % *12*
60. Create a *number* box beneath % *12*
61. Connect the outlet of % *12* to the inlet of the newly created *number* box

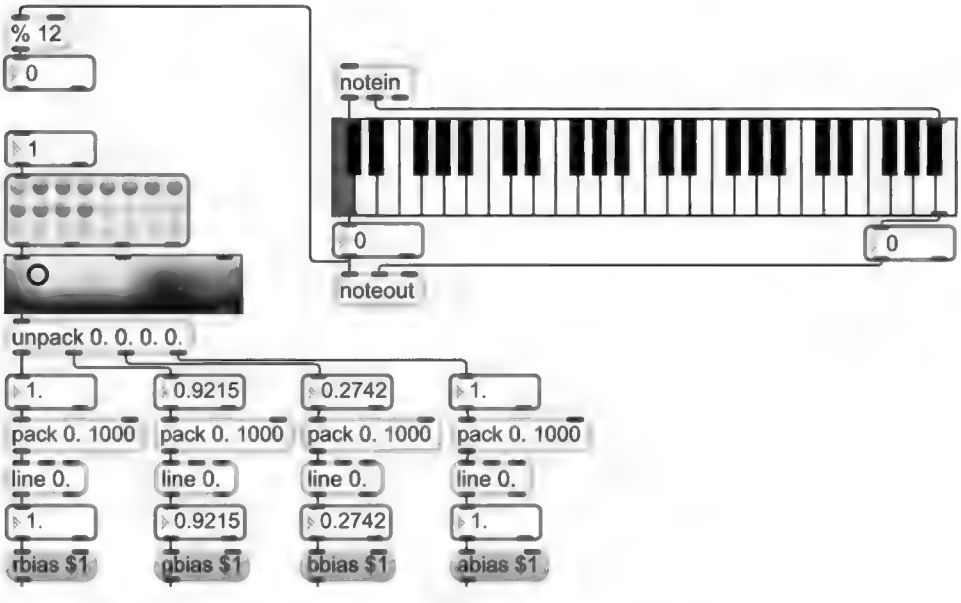


FIGURE 16.10

MIDI pitches reduced to pitch classes | color _changer.maxpat

Now, when you play the note middle C (MIDI note *60*), its pitch class value, *0*, will be sent to the *number* box connected to % *12*. It would be fine for

us to then connect that *number* box directly to *preset* except that *preset*'s first storage setting is retrieved when the number *1* is sent, not *0*. To remedy this situation, we can just add *1* to the pitch class value. After all, we're not using these pitch class values to generate any music, but simply to trigger presets.

62. Create a new object called + with the argument *1*
63. Connect the first outlet of the *number* box receiving from % *12* to the first inlet of the + *1* object
64. Connect the outlet of the + *1* object to the inlet of the *number* box connected to and directly above *preset*

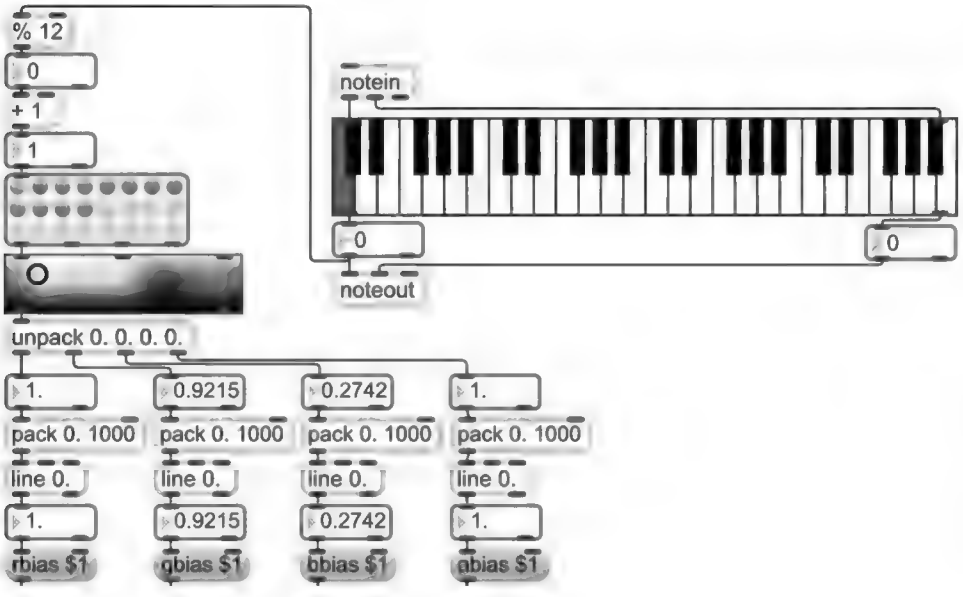


FIGURE 16.11

pitch classes are offset by 1 and used to select presets | color_changer .maxpat

When you play a note on your MIDI keyboard, it will retrieve a preset that will change the color of the camera video. While we're at it, let's change the color of *kslider*'s highlighted key to match the camera color. In *kslider*'s *Help* file, we can see that the highlighted key color can be changed by sending the *message hkeycolor* followed by 4 RGBA values. Since we already have the RGBA values from *swatch*, we only need to *prepend* that RGBA message with the word *hkeycolor*.

65. Create a new object called *prepend* with the argument *hkeycolor* above *kslider*
66. Connect the first outlet of *swatch* to the inlet of *prepend*
67. Connect the outlet of *prepend* to the first inlet of *kslider*

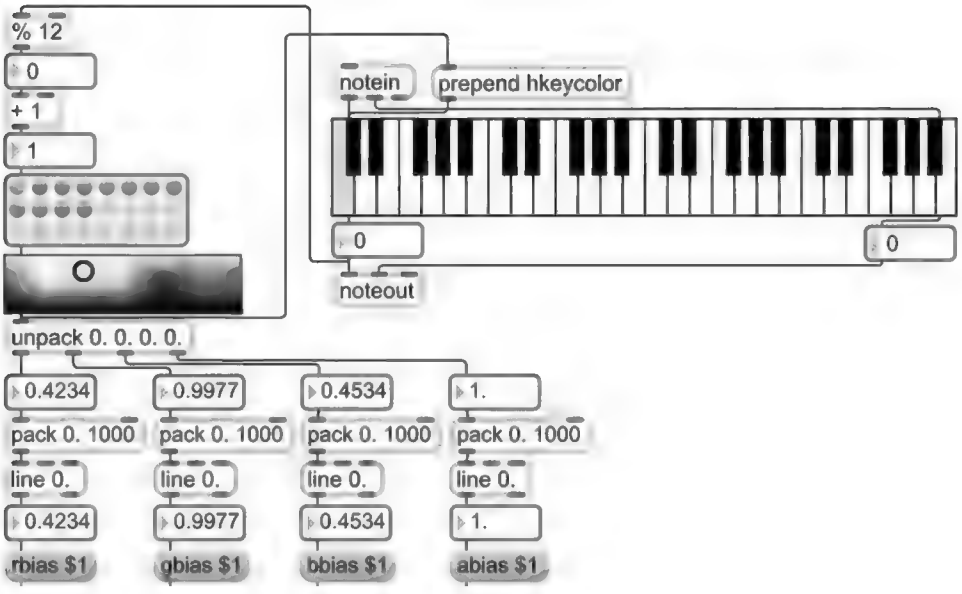


FIGURE 16.12

other elements of the patch can have their color changed as well | color _changer.maxpat

Of course, it's also possible to change the color of other things in the patch such as the background or the active preset color of the *preset* object. Think about the new and exciting ways that you can use color in performance and composition activities. Imagine using the colored noteheads in your favorite notation software with colors that matched the keyboard color and a video image. You can even make your own type of notation based on color. Activities (and even compositions and performances) that may have previously been perceived as being boring can suddenly become interesting simply because of a color tinted video image. Go figure!

In the next chapter, we will discuss using prerecorded videos inside Max as well as tracking color.

Remember:

- *Jitter* is a set of objects that handle video and visual-related content in Max.

New Objects Learned:

- *jit.qt.grab* or *jit.dx.grab*
- *jit.pwindow*
- *jit.window*
- *qmetro*
- *swatch*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Jitter Tutorials from the Help menu and read
  - Jitter—What Is a Matrix?
  - Attributes—Editing Jitter Object Parameters
  - Jitter Tutorial 5—ARGB Color
  - Jitter Tutorial 6—Adjust Color Levels
  - Jitter Tutorial 21—Working with Live Video and Audio Input

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that allows a user to see video input from a camera in at least 3 windows of varying sizes. Create controls (*slider*, *swatch*, etc.) to change the color for each window. Create some way in which sound (MIDI or audio) changes or "responds" to color.

17

Working with Video Files

In this chapter, we will work with preexisting video files located within the Max search path. Once we are able to open a video file in Max, we will create a patch that obtains some data about that video file.

By the end of this chapter, you will have created patches that detect presence in certain areas of a video. We will also examine some aspects of tracking colors in video.

Video

To load a video file in Max, we will use the *jit.qt.movie* object. The object takes width and height pixel dimensions as its arguments to specify the resolution.

1. Create a new patch
2. Create a new object called *jit.qt.movie* with the arguments *320* and *240*
3. Create a new object called *qmetro* with the argument *30* above *jit.qt.movie*
4. Connect the outlet of *qmetro* to the inlet of *jit.qt.movie*
5. Create a *toggle* above *qmetro*
6. Connect the outlet of *toggle* to the inlet of *qmetro*
7. Create a new object called *jit.pwindow* beneath *jit.qt.movie*
8. Connect the first outlet of *jit.qt.movie* to the inlet of *jit.pwindow*

9. Create 3 *message* boxes containing the text *read*, *start,* and *stop,* respectively
10. Connect the outlet of each *message* box to the inlet of *jit.qt.movie*

As you already know, the *qmetro* will send bangs to trigger the movie to play back and be displayed in the *jit .pwindow*. To read a movie file into the *jit.qt.movie* object, we would simply need to click the *read* message. To begin and stop playback we can use the *start* and *stop* messages. Since we have not yet read a movie file into the *jit.qt.movie* object, let's take a moment and look at helpful ways to locate videos and other files that are already in the Max search path.

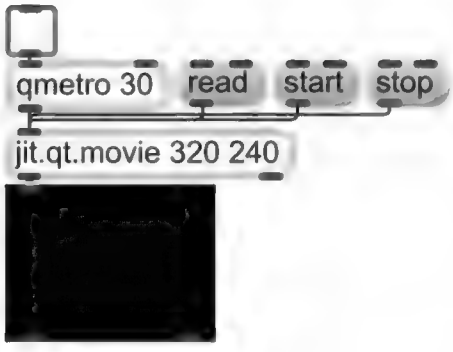


FIGURE 17.1
basic movie playback elements | presence _detection_Part1.maxpat

11. Click *File>New File Browser* from the top menu

The window that opens displays patches and other files located within the Max search path and in other folders. This provides a useful way to locate media and recently used items. Let's use the File Browser to search for video files that have the file extension *.mov*.

12. From the left side of the File Browser, select the *everything* option to display all items in the Max search path
13. In the search field, type in *.mov*
14. Locate the file *redball.mov* in the search results window. Click on the file icon and drag it onto the existing patch

Notice that a *message* box labeled *read redball.mov* appears in the patch. This message can be connected directly to the *jit.qt.movie* object. Since the movie file *redball.mov* is apparently in the Max search path, clicking the *message read redball.mov* will load the file similar to the way we loaded the audio file *jongly.aif* in a *buffer~* object.

15. Connect the outlet of the *message read redball.mov* to the inlet of *jit.qt.movie*
16. Lock the patch and turn on the *toggle*
17. Click the *message read redball.mov*
18. Use the *start* and *stop* messages to control the movie playback[1]
19. Leave the *metro* on and click the *start* message to keep the ball moving throughout the steps to follow

1. There are a two non-Jitter objects in Max capable of playing back video files: *imovie* and *movie*. The *jit .qt.movie* object, however, is much more flexible in terms of overall features.

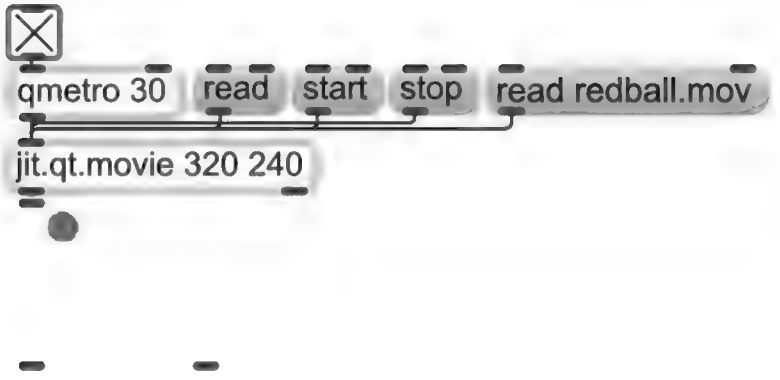


FIGURE 17.2

reading a movie located in the Max search path | presence_detection _Part1.maxpat

In this book, we've gained a lot of experience obtaining data (numbers, letters, mouse clicks, etc.) and mapping those data to other things (notes, chords, audio files, etc.). One way to get data from video is to detect motion or "presence" in a certain area of the display window. Let's build a patch that plays a different note for each different part of the patch that the red ball in the video moves to.

Presence Detection

To detect presence in a video, we need to somehow define what we want to define as "presence." For our patch with the "red ball" video, we can define presence as "when the ball enters a certain part of the display." To calculate this in Max, it would be easier for us to distinguish between tracking "color" and tracking a "difference in shading." We'll discuss color tracking later; for now, let's tell our patch to look for a difference in shading in a certain part of the display window. To make this task easier, let's remove the color altogether using the object *jit.rgb2luma* and check for different shades in a monochrome view of this movie.

Unlock your patch and

20. Create a new object called *jit.rgb2luma*
21. Connect the first outlet of *jit.qt.movie* to the inlet of *jit.rgb2luma*
22. Create a new object called *jit.pwindow* beneath *jit.rgb2luma*
23. Connect the first outlet of *jit.rgb2luma* to the inlet of *jit.pwindow*



FIGURE 17.3
color removed from video with jit.rgb2luma object | presence_detection _Part1.maxpat

Notice that the two videos are identical with the exception of color since the *jit.rgb2luma* object has converted the RGB values of the video to monochrome (black and white). Now it will be easier for us to detect the presence of the ball because we'll just have to look for differences in shading within certain parts of the display. Now let's define what "certain parts of the display" means.

We can divide up our display into several equal parts using the object *jit .scissors*. Once the display is divided, we can analyze each part separately and look for changes in shading. The *jit.scissors* object can divide the images into rows and columns (rows go across horizontally, and columns go up and down vertically). Instead of just taking arguments to specify rows and columns, *jit .scissors* takes *attributes* to specify particulars about the object.

Attributes are entered after the name of the object just like arguments and begin with the @ sign immediately followed by the name of an attribute. For *jit .scissors*, we can specify the attribute *@rows* and give it the argument *4* as well as *@columns* and give it the argument *4* to divide the display evenly into 4 rows and 4 columns. The *jit.scissors* object will then send the divided image out of its first *16* outlets (16 = 4 rows* 4 columns).

24. Create a new object called *jit.scissors* with the following attributes and arguments: *@rows 4 @columns 4*
25. Delete the *jit.pwindow* receiving from the first outlet of *jit .rgb2luma* and connect the first outlet of *rgb2luma* to the inlet of *jit.scissors*
26. Create 16 new objects called *jit.pwindow* aligned in rows of 4
27. Connect the first 16 outlets of *jit.scissors* to the 16 newly created *jit.pwindow* objects (First, connect the first 4 outlets of *jit.scissors* to the 4 *jit.pwindows* in the first row, then connect the next 4 outlets of *jit.scissors* to the 4 *jit.pwindows* in the second row, and so on)

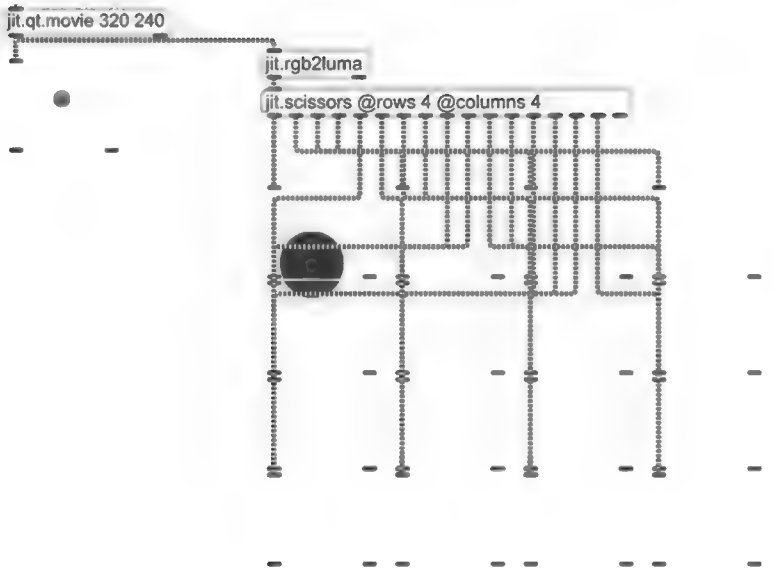


FIGURE 17.4

video image divided into 16 equal parts with jit .scissors | presence _detection_Part1.maxpat

Now that our patch has some specified areas, we can look at how much shading is in each of those areas and, if we detect a difference in shading, deduce whether the ball is in the area. The *jit.3m* object reports minimum, median, and maximum ARGB values for a matrix, so, since we've already converted our video to monochrome, we really only need to find what the median value is when the ball is not in the matrix and compare it to the value when the ball is in the matrix. Once we determine these values, we can use our existing knowledge of Max objects like *select* to cause a bang whenever specified values are received.

28. Create a new object called *jit.3m*
29. Connect the first outlet of the first *jit.pwindow* receiving from *jit.scissors* and connect it to the inlet of *jit.3m*
30. Create a *flonum* box beneath *jit.3m*
31. Connect the second outlet of *jit.3m* (the outlet that outputs the median shading value) to the inlet of the *flonum* box

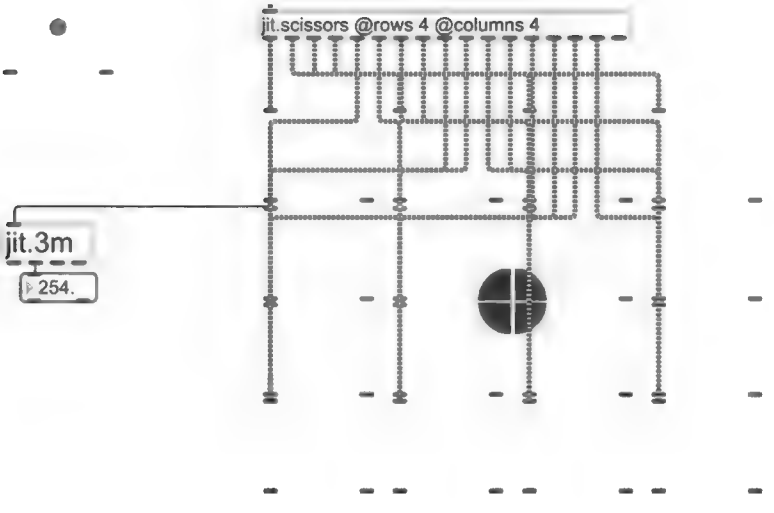


FIGURE 17.5

detect median shading value with jit.3m | presence_detection _Part1.maxpat

Notice that the *flonum* box primarily displays the value *254*, the median shading value of the white background, except when the ball enters the first *jit .pwindow*, the number drops. This makes detecting the ball easy: if the median number is less than *254*, it must be because the ball is present in some part of the display, so send a bang to trigger a MIDI note.

32. Create a new object called < (less than) with the argument *230* beneath the *flonum* box
33. Connect the first outlet of the *flonum* to the inlet of <
34. Create a *toggle* object beneath <
35. Connect the first outlet of < to the inlet of the *toggle*

When < receives a number that is less than *230*, it will send out a *1* to the *toggle*. We'll use the argument *230* instead of *254* so that our < object doesn't respond to numbers that are just a little bit less than *254*; we don't want the object to be so sensitive to numbers less than *254* that if the ball slightly appears in the *jit .pwindow*, the < object responds. Of course, on your own, you can adjust and tweak these values to achieve different results.[2]

When the ball is inside a *jit.pwindow*, the shading will be less than *230* and the < object will send out *1*s repeatedly for each median value that is below *230*. Because we don't necessarily want repeated bangs during the time when the ball is in the *jit.pwindow*, we will use the *change* object which, as you will recall, allows only one instance of a number to pass through to its outlet until a new number is received. In essence, when the ball goes into the *jit.pwindow* and its shading is less than *230*, *change* will send out a *1* until the ball leaves the *jit .pwindow*, when it will send out a *0*.

36. Create a new object called *change*
37. Connect the outlet of *toggle* to the first inlet of *change*
38. Create a new object called *sel* with argument *1*
39. Connect the first outlet of *change* to the first inlet of *sel*
40. Create a *button*
41. Connect the first outlet of *sel* to the inlet of *button*

2. *254* is obtained from the background and is a really light shade. If your background was dark and you wanted to detect a lighter shade, you would want to look for some light value greater than a dark value such as *50* (black is 0).

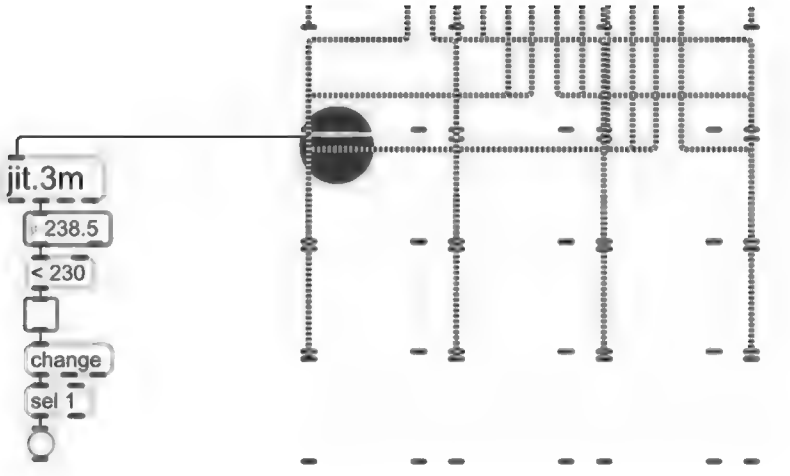


FIGURE 17.6

if shading value is less than 230, send out a 1 | presence_detection _Part1.maxpat

With your video still running (Step 19), notice that the *button* receiving from *sel* blinks when the ball enters the first *jit.pwindow*. The next step is to use this *button* to trigger a MIDI note (obviously, you can use this *button* to trigger anything you like).

42. Create a *message* box containing the number *60*
43. Connect the outlet of the *button* to the first inlet of the *message* box containing *60*
44. Create a new object called *makenote* with the arguments *100* and *300*
45. Connect the outlet of *message 60* to the first inlet of *makenote 100 300*
46. Create a new object called *noteout*
47. Connect both outlets of *makenote* to the first 2 inlets of *noteout,* respectively

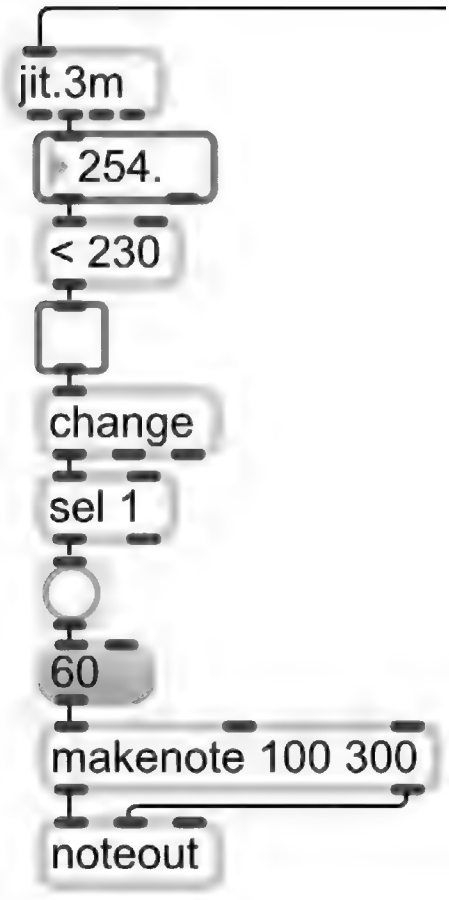


FIGURE 17.7
if shading value is less than 230, play MIDI note 60 | presence_detection_Part1.maxpat

The note middle C (MIDI note 60) will be played whenever the ball is present in the first *jit.pwindow*. Great job! Take a moment to save your patch as *presence_detection.maxpat* by clicking *File>Save*.

Here comes the laborious part: we have to do everything we did in steps 28–43 for the next 15 *jit.pwindow* objects in our patch, setting each one to a different note. It's actually not as bad as it seems: we'll just encapsulate all of the objects between and including the *jit.3m* and the *message 60* and copy the encapsulated subpatcher.

48. Highlight the objects *jit.3m*, *flonum*, *<*, *toggle*, *change*, *sel*, *button*, and the *message* box, and choose *Edit>Encapsulate* from the top menu

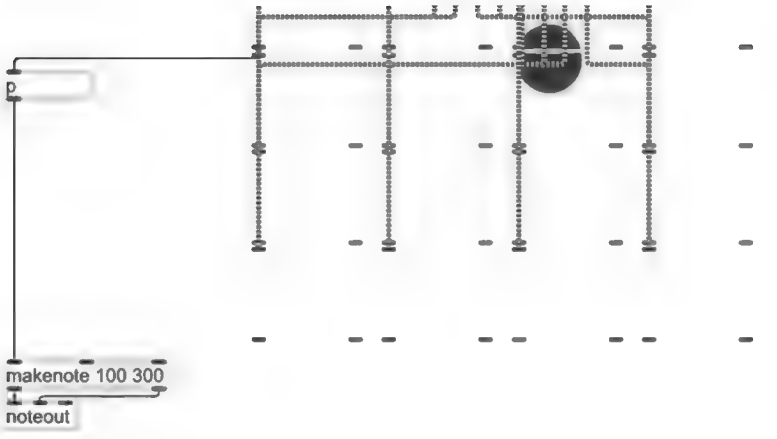


FIGURE 17.8

objects encapsulated | presence_detection _Part2.maxpat

At this point, we can simply copy the subpatcher 15 times changing the MIDI number inside the encapsulated *message* box each time. While that approach is fine, let's look at another way to do it that gives us a little bit more control.

49. Hold ⌘ (Mac) or ctrl (Windows) and double click the subpatcher (*p*)
50. Within the subpatcher window that opens, click *File>Save As* and save this subpatcher as *shade_detection.maxpat* in the same location that you saved the *presence_detection.maxpat* file

You have just created an abstraction from the subpatcher.

51. Close the subpatcher window and create a new object in the *presence_detection* patch called *shade_detection*
52. Hold ⌘ (Mac) or ctrl (Windows) and double click the subpatcher to open a window revealing that this abstraction has the same contents as the subpatcher
53. Close the *shade_detection* window
54. Ctrl+click (Mac) or right+click (Windows) the *shade_detection* abstraction and choose *Object>Open Original "shade detection"* from the contextual menu
55. Unlock the *shade_detection* patch

The two main variables in this abstraction are the amount of shading (*230*) and the *MIDI* note (*60*). Because we want to work efficiently and reuse code whenever possible, let's replace *230* with the *#1* symbol and replace MIDI note *60* with *#2*. This will allow us to specify arguments for the contents of the abstraction in the main patch next to the object name *shade_detection*.

56. Double click the < *230* object and replace the *230* with *#1*
57. Double click the *message 60* and replace the *60* with *#2*
58. Save the changes to the abstraction and close the patch



FIGURE 17.9
#1 and #2 placeholders used to provide flexibility for patch | shade _detection.maxpat

In the main patch, we can specify two arguments for the *shade_detection* abstraction. The first argument will transmit to the *#1* in the < object and the second argument will transmit to the *#2* in the *message* box.

59. Double click the *shade_detection* abstraction and enter the arguments *230* and *60* next to the object name
60. Delete the subpatcher and connect the first outlet of the first *jit.pwindow* to the inlet of *shade_detection*
61. Connect the first outlet of *shade_detection* to the first inlet of *makenote*
62. Hold ⌘ (Mac) or ctrl (Windows) and double click the *shade _detection* abstraction to reveal that the arguments *230* and *60* were sent to the placeholders *#1* and *#2*

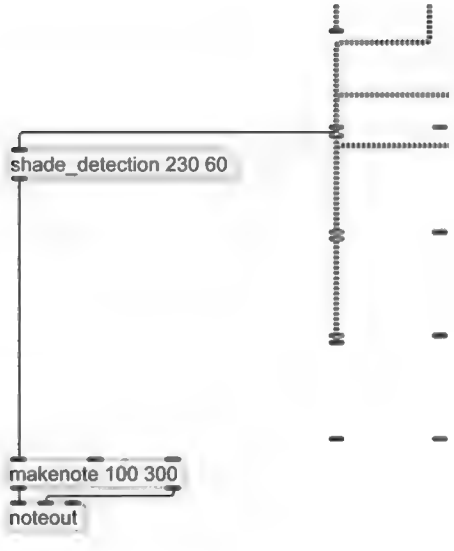


FIGURE 17.10
abstraction used with added arguments | presence_detection _Part2.maxpat

63. Copy and paste 15 instances of *shade_detection* with the arguments *230* and *60*. Align them in rows of 4
64. Connect the first outlet of each *jit.pwindow* to the inlet of each *shade_detection* (Note: be careful to ensure that you are connecting the correct *jit.pwindow* outlet to the corresponding inlet as this can, at times, be a bit confusing to look at)

65. Connect the outlet of each *shade_detection* to the first inlet of *makenote*

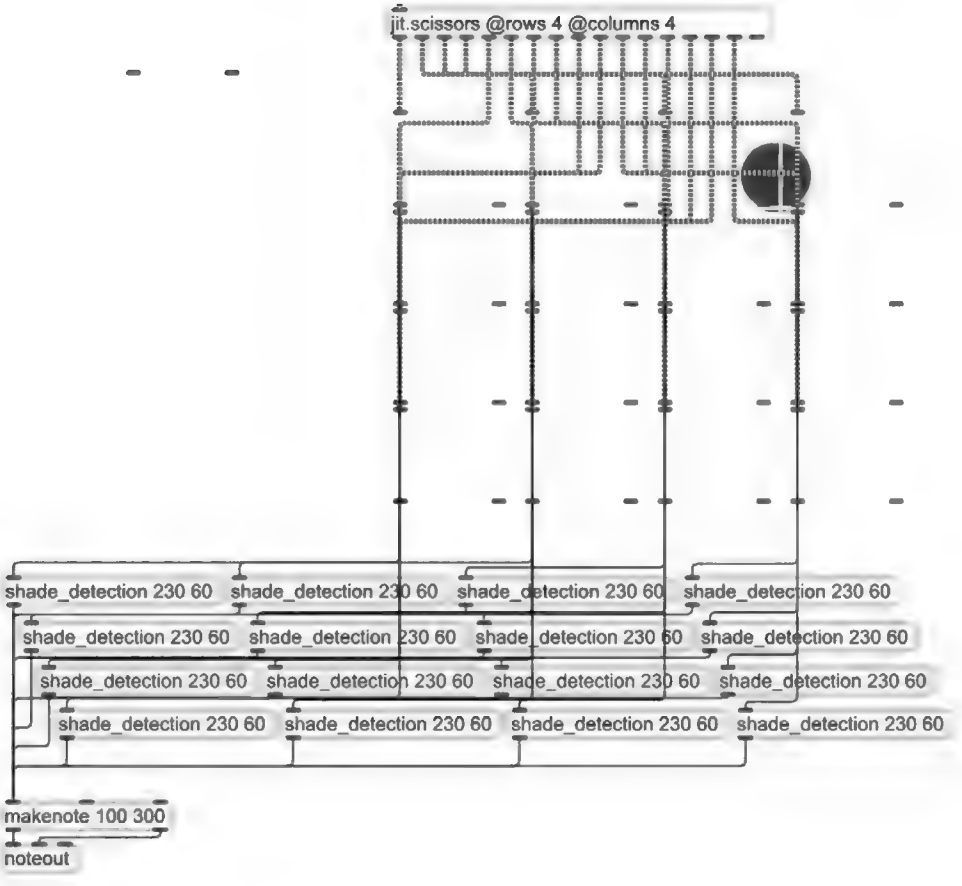


FIGURE 17.11

multiple abstractions receiving from the jit .pwindow objects | presence_detection _Part2.maxpat

Now the fun and artistic part:

66. Change the *60* in each *shade_detection* object to some other MIDI note (If you are not comfortable with equating MIDI numbers to pitches, you may use the following notes from the C major scale: 60, 62, 64, 65, 67, 69,71, 72)

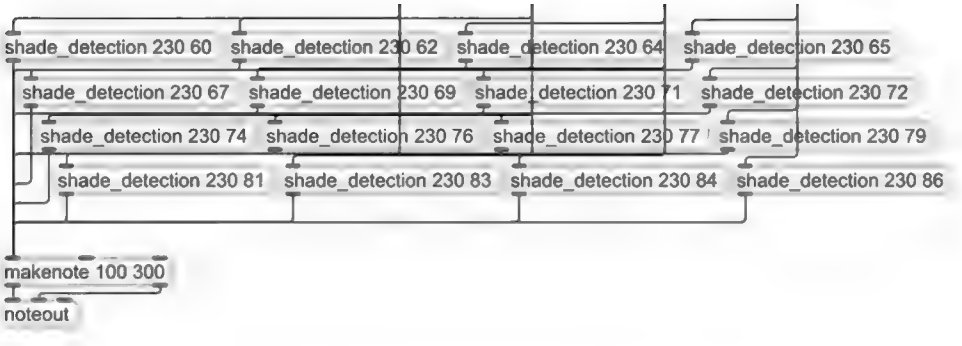


FIGURE 17.12

multiple abstractions used with different arguments | presence_detection _Part2.maxpat

You can, of course, have a fun time using live video from *jit.qt.grab* instead of a prerecorded video, thus causing the patch to function like a musical instru-

ment of sorts. As a final project, an undergraduate student of mine used real-time video with black and white index cards to create a performance instrument. I had another undergrad use a preexisting video of his roommates fighting (in good fun) in which each portion of the screen that they entered triggered a different audio file to play. The sound files included explosion sounds and excerpts from popular video games. In some strange way, the project was kind of like providing a score to a movie.

Save and close this patch.

Color Tracking

Color tracking can be a little bit more involved. Let's look at a working model by opening the file *video_color_tracking.maxpat* in the *Chapter 17 Examples* folder. Since you are likely more familiar with discussing musical ideas than video ones, just hang on and try to grasp the basic concept as we go through this section.

Upon opening the patch, you'll notice the same objects we used to play back the *redball.mov* video file in the previous patch. When you unlock the patch, you'll see that a red object that looks like a spider web is actually on top of the *jit.pwindow*; this is an object called *suckah*. The *suckah* object (yes, that's really what it's called) is used to get the color information of the pixels beneath it when clicked. In other words, if we turn the metro on, the "red ball" movie will begin playing in the *jit.pwindow,* and if we click on the ball, the RGB values of the ball will be taken by the *suckah* object. In this patch, the selected color will be set as the background color of the *panel* connected to *prepend* causing the panel to display the color currently being tracked.

1. Lock the patch and turn on the *qmetro* by clicking the *toggle*
2. Click on the red ball in the *jit.pwindow* (if you are having trouble clicking on the video while it is in motion, turn off the *qmetro* object
3. Turn off the *toggle* to *qmetro*

You likely noticed that the red color produced a square of equal size in a small drawing object below called *lcd*. The *lcd* is used simply to display the boundaries of the tracked color.

The object *jit.findbounds* is used to track the boundaries of the color. The video output of the *jit.qt.movie* object is sent to *jit.findbounds* while simultaneously *jit.findbounds* is given the minimum and maximum RGB values to look for in the video. It's these values, defined by the *suckah* object, that *jit.findbounds* will build a boundary for.

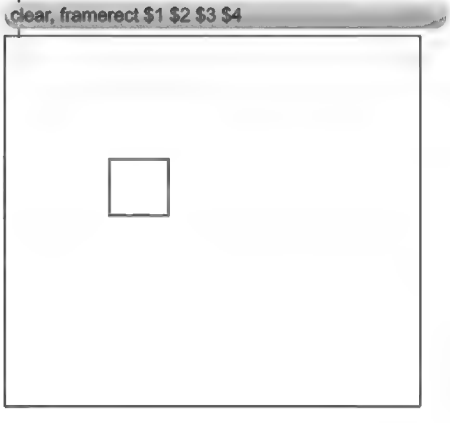


FIGURE 17.13
frame drawn in lcd object | video_color_tracking .maxpat

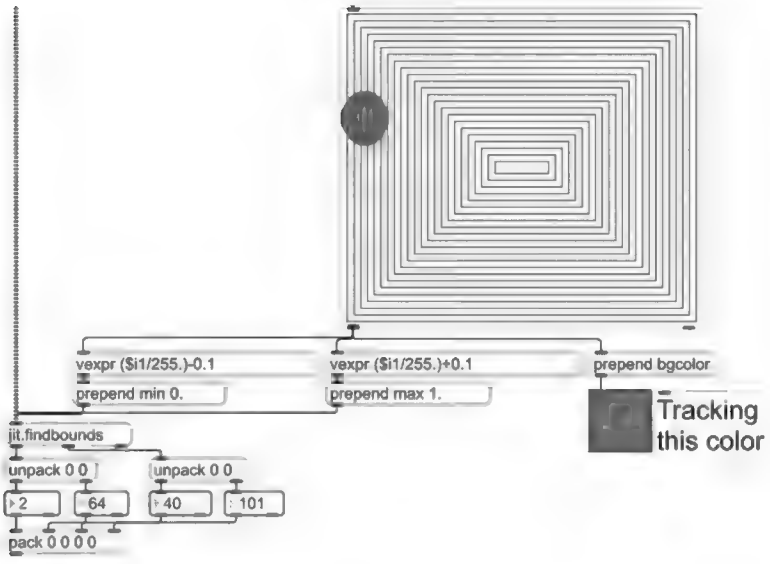


FIGURE 17.14

boundaries are detected for color presence | video_color_tracking .maxpat

The RGB values from *suckah* are outputted on a color range between *0* and *255* as opposed to *0.* and *1.* like we used with *jit.scalebias* in the previous chapter. When you click the *suckah* object, the selected RGB values of the ball displayed beneath the object are sent to two *vexpr* objects. The *vexpr* may look intimidating but is really quite similar to the *expr* object you already know. The difference is that *vexpr* will work with lists of numbers, such as our RGB values, instead of just one number. The 255 possible values for R, G, and B are divided by *255* to scale them to the range *0.* and *1.* making them suitable for use with *jit .findbounds.* The values then produce two sets of minimum and maximum color values for which *jit.findbounds* will build a boundary. A small margin of color differentia is introduced into these boundaries as a result of subtracting and adding *0.1* the values with *vexpr.*

The *jit.findbounds* then outputs two pairs of values for the minimum (left, top) and maximum (right, bottom) points from its first two outlets. These values are *unpack*ed into 4 separate values, re*pack*ed into a single message containing all 4 values. The list of all 4 values, which are essentially like dimensions, is then sent to *lcd* as part of a *clear, framerect* drawing message to visually represent the boundaries of the color being tracked.

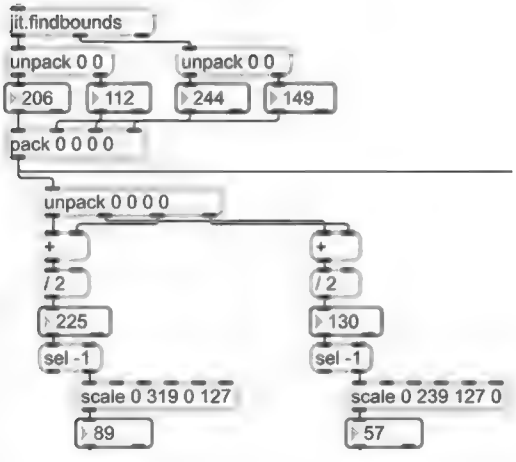


FIGURE 17.15

boundaries dimensions are scaled to MIDI pitch range | video_color _tracking.maxpat

Following the dataflow from *pack,* the dimension values are also then *unpack*ed. Since we now have the outlying borders of the red ball, it would be ideal to get the center point of the ball. To get the center point, we simply add the left and right points and divide them in half, and also add the top and bottom points and divide them in half. We are

then left with two coordinate values: a horizontal position and a vertical position.

In this patch, the maximum possible values for horizontal and vertical position are 320 and 240. With the use of the *scale* object, the horizontal position values *0* to *319* are scaled to MIDI pitches *0* to *127*. The vertical position values *0* to *239* are scaled to MIDI velocities *0* to *127*. This is similar to the way we used the *mousestate* object to turn the position of the mouse on the screen into pitches and velocities in Chapter 8. The *change* object is used to filter out repetitions of the same note.

Of course, at this point, you can use your knowledge of pitch filtering to a diatonic key to move this patch away from atonality. You can also use a camera input instead of a video. There is a functional color-tracking patch in the EAMIR SDK called *EAMIR_Camera_Control* (accessible from *Extras>EAMIR.org* in the *Examples* menu).

Preassembled Video Patches

In my experience, video concepts tend to be more foreign to beginner Max users who are musicians than the MIDI concepts introduced in earlier chapters. Cycling '74 has made working with video a lot easier by developing "Vizzie," a collection of *bpatcher*s that demonstrate video processing modules of code similar to the way that we used the EAMIR *bpatcher*s in previous chapters. To access these *bpatcher*s, ctrl+click (Mac) or right click (Windows) a blank portion of the patch and select *Paste From> VIZZIE-CLIPPINGS*. There are many different modules that you can explore by simply clicking a module name from the menu. When the module has copied into your patch, you may hold your mouse over the name of it to reveal a short description. As mentioned in Chapter 6, customized shortcuts like this can be added to this menu by copying patches to the *patches>clippings* folder inside the Max application folder.

Remember:

- The File Browser displays patches and other files located within the Max search path and in other folders.
- Attributes are used to specify particulars about an object and are entered after the name of the object just like arguments and begin with the @ sign immediately followed by the name of an attribute.

New Objects Learned:

- *jit.qt.movie*
- *imovie*
- *movie*
- *jit.rgb2luma*
- *jit.scissors*

- *suckah*
- *lcd*
- *jit.findbounds*
- *vexpr*

Additional Tutorials:

Tutorials are available from within the Max application by clicking Help from the top menu.

- Select Jitter Tutorials from the Help menu and read
  - Jitter Tutorial 4—Controlling Movie Playback
  - Jitter Tutorial 13—Scissors and Glue
  - Jitter Tutorial 25—Tracking the Position of a Color in a Movie
  - Mouse Drawing—Introduction to Drawing
  - Random Drawing—Working with Random Numbers
  - Procedural Drawing—Creating Procedural Code

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that tracks color or presence and uses the tracked value to do something interesting with music. You can design a musical instrument that plays back pitches or prerecorded sounds. You can make a series of looping audio files and use presence detection to trigger each audio file, or modify the volume of each audio file.
- Explore the many great modules in Vizzie and create a patch that does something interesting with video.

18

Video Research Instrument

In the next few chapters, we will examine some larger projects covering a number of topics in music education. In this chapter, we will look at a research instrument designed to measure the time when a participant responds to some stimuli while watching videos.

Stimulus Testing Instrument

1. Open the file *STI.maxpat* from the *Chapter 18 Examples* folder

This patch allows a researcher to show participants one or more video files and ask them to press the space bar in response to whatever the researcher wants to observe. For example, the researcher might show a video of an orchestra performing Samuel Barber's *Adagio for Strings* and ask participants to press the space bar whenever they experienced "thrills" or "chills." The program would then record the exact times, with respect to the video, that each participant experienced such a reaction. The researcher could then compare the times from that participant's session to the times from another participant's session.

Let's walk through the process together, though, unfortunately, instead of seeing an orchestra, you'll have to endure a homemade movie from my trip to Colorado (it's short—don't worry). While you are watching the video, pretend that you are experiencing "thrills" and press the space bar a few times. To start the video, press the *Return/Enter* key; this will display the video in fullscreen.

When the video comes to an end, press the *esc* key to close the fullscreen view. There's not much audio in this video, but you can click the "test audio" *button* to ensure that your sound card is working (the *ezdac~* should be "on" by default with the icon colored red).

2. Press the *Return/Enter* key to start the video in fullscreen view. Don't forget to press the space bar a few times in "response" to the video stimulus
3. Press the *esc* key when the video is finished to exit fullscreen view
4. Click on the *message* box containing the text "View Data" to display a text file containing the hour, minute, second, and millisecond values for each time you pressed the space bar[1]
5. Close the window and click the *message* box containing the text "Save Data as Text"
6. In the *Save* dialog box that opens, enter the file name *results.txt* and click *Save* to save the data as a text file on your computer

Let's take a look under the hood and examine how this patch works. Although the patch may seem a bit complex, as you'll see, it's rather simple.

7. Unlock the patch and exit Presentation mode (you may need to zoom out to see all the objects in the patch)

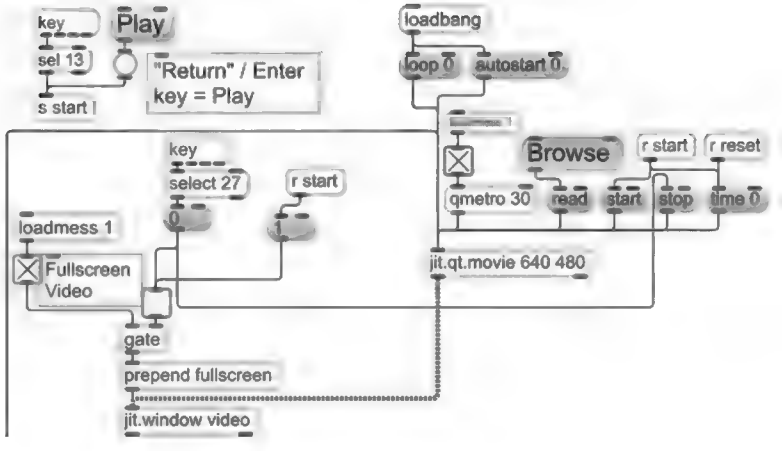


FIGURE 18.1

basic movie playback elements (right) with fullscreen components (left) | STI.maxpat

You've probably already figured out that the key controls work by using *key* and *select* objects. You will see *key* and *select* objects working in conjunction with the *send* (or *s*) and *receive* (or *r*) objects throughout this patch. The top left portion of the patch shows the basic objects for playing back video.

The basic commands you would expect to send to *jit.qt.movie* (*read, start,* stop) are present. Notice that the *browse message* box, a message that *jit.qt.movie*

1. For more information on the format of the data, click the *message* box containing the text "Data Format."

doesn't understand, simply triggers the *read* message, a message that *jit.qt.movie* does understand. When the *message time 0* is sent to *jit.qt.movie*, the loaded video will automatically return to its starting point at the beginning of the file. Notice that the *time 0* message receives a bang from the *r* objects named *start* and *reset*. We will see, in other portions of the patch, the related *s* objects named *start* and *reset* that communicate with these *r* objects. When the patch loads, *loadbang* will ensure that the video does not loop automatically.

You will notice the *message* box containing the word *Play* and the *key/sel* object combination to its left. These objects allow you to press the *Play message* or press the return/enter key on your ASCII keyboard (ASCII number *13*) to send a bang to the *s* object named *start*. We just saw that the related *r start* object is connected to the *message* boxes *start* and *time 0* that are connected to *jit.qt.movie*.

Instead of a *jit.pwindow* object, *jit.qt.movie* sends the video signal to the *jit.window* object, which displays the video in a floating window. The *jit.window* object has been given the name *video,* which in turn became the name of the floating window. You will see the name *video* displayed in the top menu bar of the video window which shows that the *jit.window* object named *video* is related to the floating window named *video*.

The floating window can be set to display the video in fullscreen view by sending the *jit.window* the *message fullscreen 1*. For the purpose of this patch, I wanted the video to automatically be entered into fullscreen view as soon as the user clicks the *play* message or presses the *enter/return* key. However, I also wanted to build in the option to not let the video be displayed in fullscreen automatically, as some researchers may prefer the video to be open onscreen in a window alongside other stimuli. To allow for these options, a simple *gate* is used to circumvent a *toggle* to the *prepend fullscreen* object. By default, a *loadmess 1* is used to open and display the *gate* as being able to send out *1* and *0* from the *toggle* connected to *gate*'s right inlet. These numbers will be sent to the *prepend* object where the *message fullscreen* will be placed in front of them. Thus, *jit .window* will receive the *fullscreen 1* message when the *toggle* is checked on.

An *r start* object is used to send a *1* to this *toggle*. A second *key/select* combination is set to send *toggle* a *0* whenever the *esc* key (ASCII number *27*) is pressed. This ultimately sends *jit.window* the *fullscreen 0* message when the *esc* key is pressed. Note that this key also triggers the *message stop* to be sent to *jit .qt.movie*.

The next interesting bit of the patch is the part that creates the "timestamp" each time the space bar is pressed. This is done with a simple *clocker* object that counts milliseconds and the *translate* object that converts milliseconds into hours,

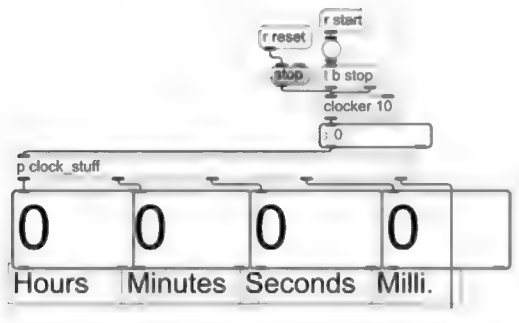


FIGURE 18.2
basic stopwatch-style clock | STI.maxpat

minutes, seconds, and milliseconds. Notice the *r start* at the top of the patch sends a bang to *t* (or *trigger*) which first sends the message *stop* to *clocker* (in case *clocker* was running) followed by a *b* (for bang) to start the *clocker* counting every *10* milliseconds. When the video begins playing, *clocker* begins counting. The millisecond output of *clocker* is sent to a small *subpatcher* I've articulately named *clock_stuff*.

8. Hold ⌘ (Mac) or ctrl (Windows) and double click the subpatcher *clock_stuff* to view its contents in a new window

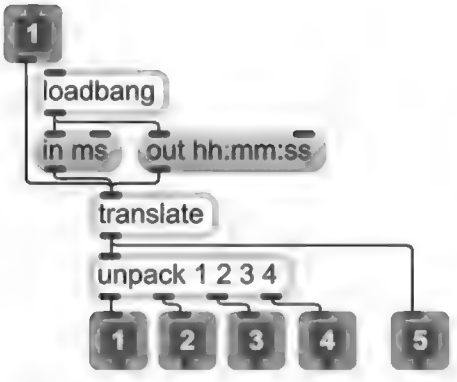


FIGURE 18.3

translate object converts milliseconds to hours, minutes, and seconds | clock_stuff subpatcher in STI.maxpat

The milliseconds are sent to an object called *translate,* which has been told to take incoming milliseconds (*in ms*) and output hours, minutes, and seconds (*out hh:mm:ss*). The outlet of *translate* sends out a single list containing hours, minutes, seconds, and milliseconds to *unpack*, which separates each element to the first 4 *outlet* objects. These first 4 outlets are used to display the time in the *number* boxes of the main patch. The complete list is also sent to the fifth outlet. As we will see in a few moments, it is the list sent to this fifth outlet that is used for the "timestamp."

9. Close the subpatcher window

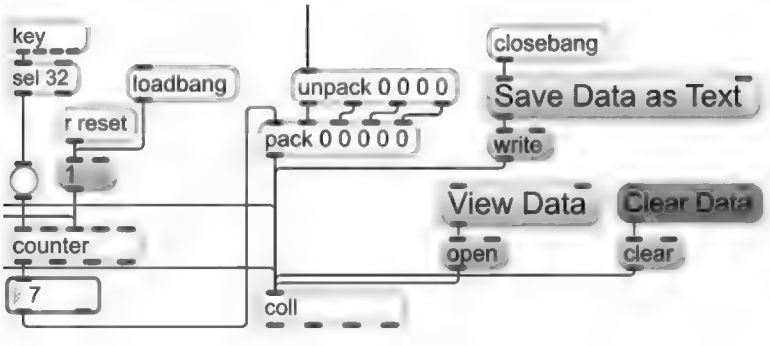


FIGURE 18.4

clock time is packed with incrementing index values in a coll each time the space bar is pressed | STI.maxpat

The complete time message is sent from the fifth outlet of the subpatcher to another *unpack* object. Here, each time value is u*npack*ed and re*pack*ed into a list of 5 elements. The first element of this new list is to be determined by the *counter* object. Each time someone presses the space bar (*key*/*sel 32*) a bang is sent to the *counter* object. The number from *counter* is packed as the first element of a list containing the 4 time elements. However, when this new list is sent from *pack* to *coll*, the *counter* number becomes the *index* (address) of the *elements* that are the time values. Thus, each time the space bar is pressed, a new index is generated from *counter* with the associated time values of when the

space bar was pressed as the elements for that index. Since the *counter* begins counting as soon as the video starts, the time values are recorded in *coll* with respect to the time in the video. The other colorful *message* boxes in this section of the patch simply trigger *messages* to *coll* such as *open* to open the *coll* in order to view the data inside. The *write* message allows users to save the *coll* data as a text file. Notice that the *closebang* object, which, as the name suggests, sends a bang when the patch is closed, is used to ensure that even if your research participant accidentally closes the patch after watching the video, you are still prompted to save the data before they are lost.

In fact, you will see a lot of "foolproofing" techniques in this patch such as *loadbang* and *loadmess* objects. In addition, *s* and *r* objects named *reset* are used to allow researchers to click one button to reset some object such as *clocker* and *counter* to their default values without having to close and restart the patch.

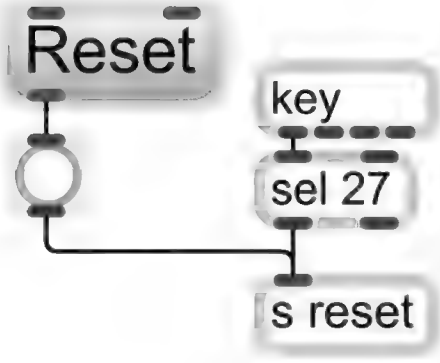


FIGURE 18.5

reset button is used to clear data and restore default parameters | STI.maxpat

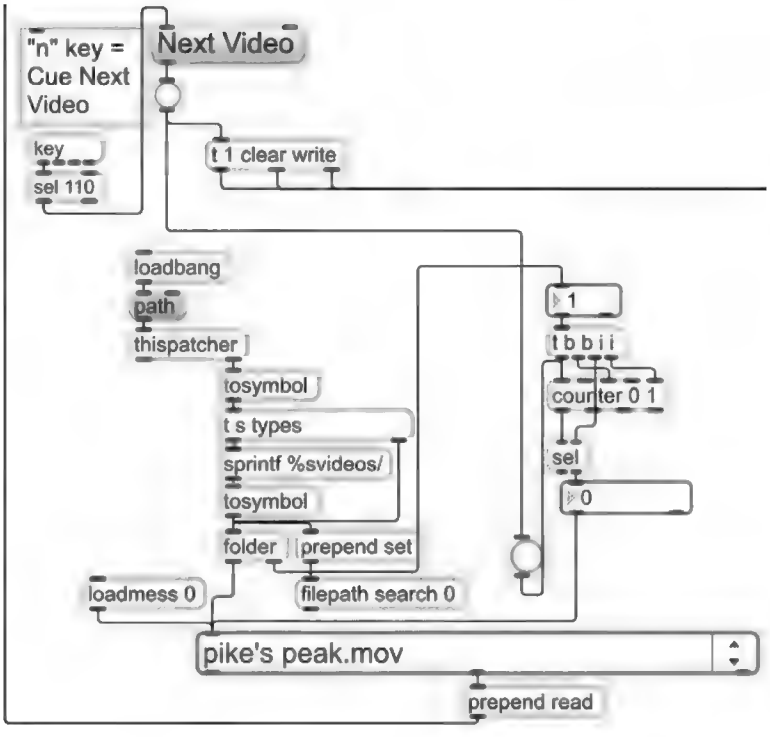


FIGURE 18.6

entity allows videos to be read from a single directory and used in succession | STI.maxpat

Working with Paths

In the bottom left portion of the patch, you will see another *key/sel* combination that allows the *n* key or the *message* box containg the text *Next Video* to be used in instances when the researcher would like to use more than one video in his

or her study. When the *message* is clicked or the *n* key is pressed, a bang is sent to a *trigger* that first sends the *write* message to *coll* which prompts the user to save the results for that video's session. Next, the *clear* message is sent to clear the data in the *coll* in preparation for the next video. Then, the number *1* is sent to reset *counter* to the number *1* so that the *coll* indexes begin, once again, at *1*. In addition, a bang is sent from a *button* to a *counter/sel* combination that reads through each video in the *umenu*. As you can see, the name of the file in the *umenu* is *prepend*ed with the word *read* and sent to the *jit.qt.movie* object allowing the video to be played. The way in which filenames make their way in the *umenu* is a little bit more involved than the rest of the patch.

This next little part of the patch is something I learned from composer and programmer Luke Dubois, who coauthored Jitter and, incidentally, was the professor who introduced me to Max when I was in graduate school. The goal of this section of the patch is to allow the researcher to load a bunch of video files into a single folder, have all of the videos load into Max for playback with *jit.qt.movie*, and have the videos play back sequentially without having to reload a new video each time.

As you may have noticed when you opened the *STI.maxpat* file, there is a folder inside the *Chapter 18 Examples* folder called *videos*. Inside this folder is currently the video "*pike's peak.mov,*" the video that loads by default when the patch loads and is displayed in the *umenu*. However, if you place more videos inside this folder and click the *path* message, those videos will be displayed in the *umenu* as well and will be cued after the first movie has finished playing.

10. In the *Chapter 18 Examples* folder, move the file "garden of the gods.mov" into the *videos* folder (Note: this video has no audio track)
11. Lock the patch and click the *message* box containing the word *path* in the lower left portion of the patch

This part of the patch not only lists all the files within the folder in a *umenu* for loading by *jit.qt.movie* but also remedies issues sometimes caused when you try to load files that have spaces in their name. Brace yourself for this next part.

The *message path*, when sent to *thispatcher*, yields the full file path name of the open Max patch *STI.maxpat*. This file path will be some location on your computer where the patch currently resides (for example, "*HD:/Users/Documents/patches/Ch 18 Examples/*"). The file path is then converted to a symbol with the *tosymbol* object. The *t* object then receives the file path symbol and sends out the word *types* from its second outlet to the *folder* object below while the file path is sent from *t*'s first outlet. The file path is then sent to the *sprintf* object.

The *sprintf* object can be a little scary to use at first because some of the object's functionality refers to the programming language *C*. For our purposes, we are going to use *sprintf* to take the file path of our Max patch and append the

name of the folder where we are going to store all our videos; in this case, I named the folder *videos*. The argument for *sprintf* is

%svideos/

in which *%s* is a placeholder for a symbol sent to *sprintf*'s inlet. The incoming symbol to *print* will be the file path of the Max patch (*STI.maxpat*):

"*HD:/Users/Documents/patches/Ch 18 Examples/*"

The path returned from the outlet of *sprintf* became the following path which refers to the folder where are videos will be placed:

HD:/Users/Documents/patches/Ch 18 Examples/videos

The newly formatted path was returned from *sprintf* as a list message lacking quotes. To convert the message back into a symbol, the message was sent to a *tosymbol* object where it is formatted as "*HD:/Users/Documents/patches/Ch 18 Examples/videos.*" Almost done; stay with me!

This formatted file path symbol is then sent to the *folder* object that lists certain files within the specified folder. By default, the *folder* object looks for text files and Max files within the folder. You can specify file extensions to look for by sending *folder* the message *types* followed by a file extension (such as PICT, or AIFF, for example). However, if just the message *types* is received by *folder*, all the files in a folder will be displayed regardless of file extension. As you'll recall, the *t* object sent the *types* message out of its second outlet to the *folder* object before *t* passed the file path symbol out of its first outlet.

Therefore, the names of all of the files within the *videos* folder will be sent from *folder*'s outlet and displayed in the *umenu* object. Selecting one of these file names, such as *pike's peak.mov*, from the *umenu* will send the file name to the *prepend read* object, which will send the message *read pike's peak.mov* to *jit .qt.movie*. However, the files are not in the search path of the Max patch because they are located within a folder alongside the Max patch. The files are also not part of the Max search path like the *redball.mov* file, so sending *jit.qt.movie* the name of the movie file isn't going to load it. We need a way to temporarily set the *videos* folder into the Max search path so that we can select the videos from the *umenu* and have them load by *jit.qt.movie*.

This is where the *filepath* object comes into action. As you can see, the full file path is sent from *tosymbol* to *prepend set* which creates the following *message*:

set "*HD:/Users/Documents/patches/Ch 18 Examples/videos*"

The *filepath* object is used to place additional search paths in the Max search path as we did when we created a folder for our external objects called *My Max*

Objects in Chapter 6. Giving the *filepath* object the argument *search 4* would set the path above into the Max search path and save the location into slot number *4* beneath the "My Max Objects" slot. If you give *filepath* the argument *search 0*, however, the location of the folder is only temporarily stored in the Max search path until Max closes; the location is not stored in one of the search path slots. This allows you to reference the video files in the *videos* folder by their names since those files are then in the Max search path, albeit temporarily.

The next part of the patch describes how the *Next* message scrolls through each video in the *umenu*. The number of items within the folder is sent from *folder*'s right outlet to the *t* object with the arguments *b b i i*. The number is passed to the *counter* object to set the maximum number that *counter* should count to. The same number is sent to *sel* to give it an argument.

When the *Next* message is sent by clicking in the *message* box or pressing the *n* key, a bang is sent to increase the *counter*. Since the maximum value has been reset to the number of files within the *videos* file path, the *counter* object will increase in numbers causing the name of the corresponding item in *umenu* to be sent to *jit.qt.movie*. The *sel* object is used to filter out the maximum value of *counter*, received in its right inlet as its argument, which is one number higher than *umenu* can understand given that *umenu* refers to its items starting with the number *0*.

Any remaining parts of the patch are either self-explanatory or things we have already covered. Close the patch.

Application

Although I've been referring to this patch in the context of a research project, it would be valuable in a number of different contexts. It can be used for assessements—for example, videos of different musical excerpts are loaded and the students must identify certain musical events, such as when the modulation occurs or when the tempo seems to increase. In fact, with a little tweaking, you can get rid of the video component altogether and simply use it for audio purposes such as music excerpt listening quizzes or even hearing tests.

New Objects Learned:

- *jit.window*
- *clocker*
- *translate*
- *closebang*
- *folder*
- *sprintf*
- *filepath*

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch similar to the one we just discussed that uses *sfplay~* to play back a collection of *.aif* or *.wav* files from a single folder. Design the patch so that a user is asked to press the space bar at a certain time in the track (during a modulation, within 30 seconds of when the oboe enters, when the recapitulation has ended, etc.). If the user presses the space bar within the correct time region, cause the patch to play some sort of "correct" sound such as a chime or a bell. If the user press the space bar at the incorrect time or not at all after the window of time has expired, cause the patch to play some sort of "incorrect" sound such as a buzzer or an explosion (yes, an explosion).

19

Informal Music Learning Instruments

In 2010, I was involved in a research project called the *Interactive Music Technology Curriculum Project* (Manzo & Dammers, 2010), or *IMTCP;* its goal was to teach music composition and performance to students who have no musical training by using software instruments that allow them to play chord functions.[1] In this chapter, we will examine the patches developed for this project. As you will see, even though each patch facilitates a different musical activity, the patches themselves use similar chunks of code neatly organized in *bpatcher*s.

E001

These patches allow students to play chords, at first, with the number keys (1–8) from a computer keyboard as they learn about diatonic scale degrees and chord functions. Students in this project were asked to go online and get the YouTube links to their 10 favorite songs. The faculty for this project took those songs and reduced each part of the form (verse, chorus, bridge, etc.) to a set of numerical chord functions within a key. For example, the verse would be referred to as a

1. The project was influenced by the Manhattanville Music Curriculum Project (MMCP; Thomas, 1970) and Lucy Green's research on Informal Music Learning (Green, 2008).

"1 5 6 4" in C Major while the chorus was a "2 4 1 5." Students would then use a patch to play back the chord functions using the ASCII keyboard.

1. Open the file *E001.maxpat* from within the folder *E001* located in the *Chapter 19 Examples* folder[2]
2. Press the number keys on your computer keyboard (1–8) to play back the C Major scale. Use the space bar to end the sustain
3. Click the *toggles 3* and *5* to add a third and a fifth to the output. Notice that the name of the triad is now displayed on the right

We've looked at patches similar to this in the EAMIR SDK. Let's take a look at some of the more interesting and novel features of this patch.

4. Unlock the patch and exit Presentation mode (you may need to zoom out to see all of the objects in the patch)

Several of the *bpatcher*s from the EAMIR SDK comprise this patch. As a result, discussing most of this patch's functionality would be a review. It is worth pointing out the small chord analysis portion of the patch at the lower right that uses the *modal_chord_analysis* abstraction (from the Modal Object Library)—not because of the way the abstraction functions but by the way in which the data are revealed.

The abstraction calls for at least 3 notes to be present for it to be analyzed properly, so there is no need to display the chord analysis parts of the patch in Presentation mode when there are only two chord tones present because no analysis is possible. Having seemingly purposeless objects visibly present in the patch could be confusing to the users. What we really need is a way to hide these objects while there are only two chord tones present and display the objects again when at least 3 chord tones are present.

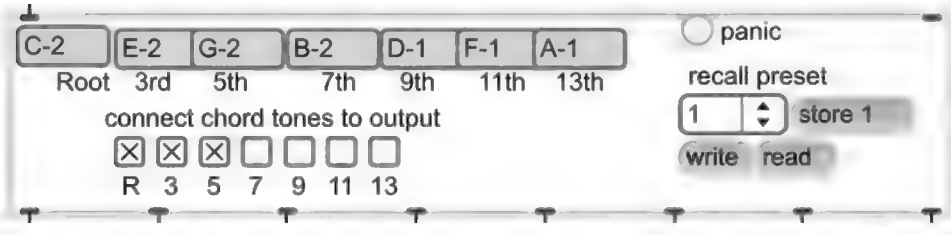


FIGURE 19.1
bpatcher to control chord tones | E001.maxpat

In this patch, the "chord generation" *bpatcher* controls the total number of notes present in each chord through 7 *toggles*.[3] These notes, which are still just pitch classes, are then outputted to the "chord voicing" *bpatcher* where they are voiced at different octaves. Just before the pitch classes reach the "chord

2. Notice the *eamir_mol_triad.xml* also in the folder which contains chord presets saved from using the *pattr* object.

3. As you'll recall from Chapter 7, this *bpatcher* uses *gate* objects to control the output of chord tones.

voicing" *bpatcher*, they are also sent to *thresh* and the *modal_chord_analysis* abstraction.

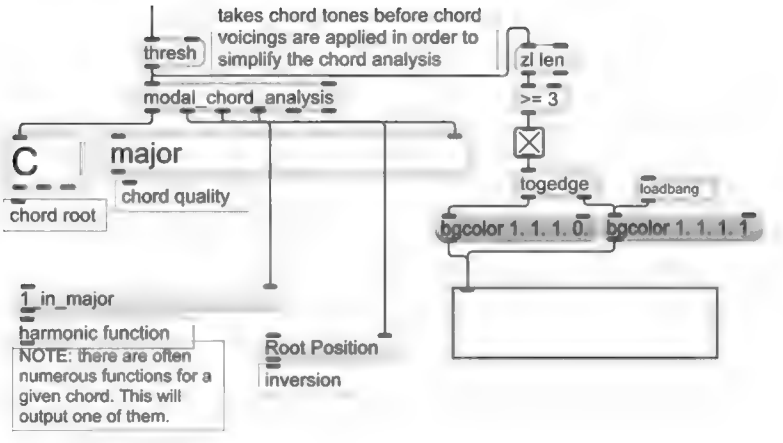


FIGURE 19.2

modal_chord_analysis abstraction reveals information about chords, but is hidden when less than 3 chord tones are present | E001.maxpat

Using *zl len*, the length of the list of chord tones is compared to the >= (greater than or equal to) object. If the number of chord tones is *3* or more, the *toggle* will become checked. The *togedge* object is used to report *0* to *1* (or *unchecked* to *checked*) transitions from the *toggle*. When the *toggle* is checked, a bang is sent from the first outlet of *togedge* to the *message* box containing the message *bgcolor 1. 1. 1. 0.* (a white color with an alpha value of *0* which makes the color transparent) which causes the background color of the *panel* object below to change from its default white background color (*bgcolor 1. 1. 1. 1.*) to a transparent background.

In Presentation mode, the *panel* object is placed on top of the *message* boxes that display the chord analysis information in order to cover them while only one or two chord tones are in use. When 3 or more chord tones are in use, the *panel* becomes transparent and the *message* boxes are visible.[4]

E005

Let's close this patch and look at a different version with more features. As the project progressed and students became more familiar with diatonic scale degrees/chord functions, traditional instruments like keyboards and guitars were introduced. Students were taught to play a major scale on these instruments. As they played each note of the scale, a Max patch was used to play additional notes in order to build a diatonic chord using each scale degree the student played as the root of the sounding chord.

4. The message *presentation $1* can also be used to add or remove an object from Presentation mode, but the position of the object in the patch will fluctuate when this is done.

1. Open the file *E005.maxpat* from within the folder *E005* located in the *Chapter 19 Examples* folder
2. Unlock the patch and exit Presentation mode (you may wish to zoom out of the patch to see all the objects)

This patch is almost identical to the *E001.maxpat* file we looked at earlier, but it has the addition of a MIDI pitch-tracking subpatcher just like the one we discussed in Chapter 13. Students can have chord tones added to the scale degree they play by way of a built-in microphone or MIDI keyboard. The sound card's built-in line input can also be used with electric guitars or external microphones/pickups on acoustic instruments.

The idea behind using this patch was to treat each note of the scale as a trigger to play a diatonic chord just as the number keys were used to trigger diatonic chords. The software could be removed when students were able to play the remaining chord tones on their instrument—sort of like using "training wheels" when learning to ride a bike.

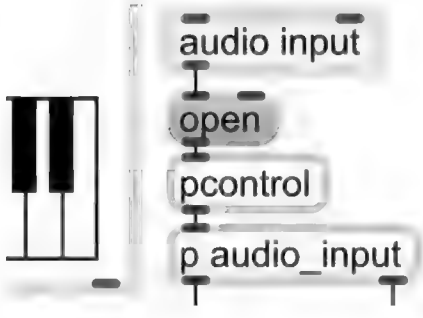


FIGURE 19.3
audio_input subpatcher encloses reveals several audio bpatchers previously discussed | E005 .maxpat

Close this patch and let's look at another patch in this series.

E003

1. Open the file *E003.maxpat* from within the folder *E003* located in the *Chapter 19 Examples* folder

This patch deals primarily with chord progressions. Students are asked to enter a set of diatonic chord functions from progressions they have encountered in their favorite popular music. The patch will play back the chord progression at a user-defined tempo, key, timbre, and playback style from which the student may sing the song using the patch as accompaniment, improvise a melody, make changes to the key and mode, or perform many other tasks.

2. Enter the numbers *1 5 3 4* (note the space between each number) in the *textedit* box at the top and press the *return/enter* key. Notice that the *1 5 3 4* progression has been added to the progressions list (*jit.cellblock*)
3. Click the *start/stop toggle* to start the progression playback
4. From the *umenu* labeled *playback style,* change the playback style from *sustained* to *style 1* and click the quarter note symbol to the right
5. Increase or decrease the tempo, change the chord tones, change the timbre, and make other changes to this performance

6. Add another progression (such as *6 4 1* 5) by clicking in the *jit .cellblock* cell beneath the *1 5 3 4* and entering new numbers in the *textedit* box and pressing the *return/enter* key. Both progressions can be toggled for playback by clicking on the cell in *jit .cellblock*
7. Click the *start/stop toggle* to stop the progression playback

In this patch, you can load up the intro, verse, chorus, bridge, and coda sections of a song and sing on top of them while the patch provides the harmonic accompaniment. It makes for a nice way to discuss form. Let's take a look inside the patch.

8. Unlock the patch and exit Presentation mode (you may wish to make the patch window larger to see all the objects)

This patch is similar to the last two that we looked at except that instead of playing the chord functions by pressing a key, a chord progression is taken and each chord in the progression is played sequentially.

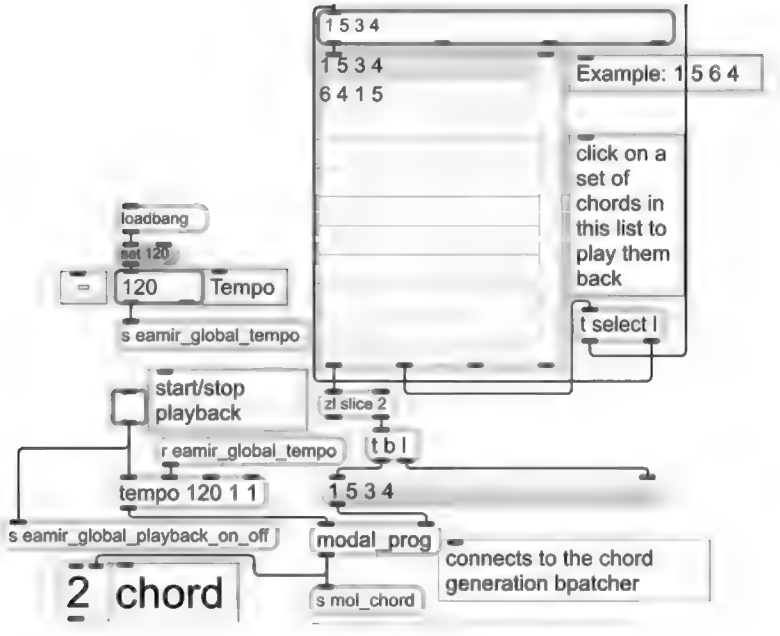


FIGURE 19.4

chord progressions entered in jit.cellblock are sent to the modal_prog abstraction | E003.maxpat

As you can see in Figure 19.4, taken from the bottom center portion of the patch, the numbers entered in the *textedit* box are sent to the cells in the *jit.cellblock* object. When a cell in the *jit.cellblock* object is selected, a list is sent from *jit.cellblock*'s first outlet containing the *column number, row number*, and the *value* (contents) of the cell. For the first cell, in column *0* and row *0*, containing the numbers *1 5 3 4*, the message sent from *jit.cellblock*'s first outlet would look something like this: *0 0 1 5 3 4*.

For this patch, we don't really care about the column and row numbers, so we will slice those two values from the front of the list using the object *zl slice*

with the argument *2*. This leaves us with just the progression, which is sent to the abstraction *modal_prog* discussed in Chapter 9. The output of the *tempo* object is then used to move through each chord in the progression sending the chord number to the *s mol_chord* object which is received (*r mol_chord*) by the "chord generation" *bpatcher*.

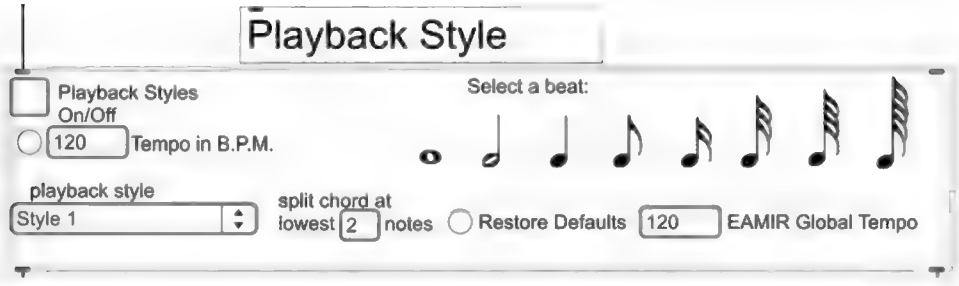


FIGURE 19.5

bpatcher allows playback in different style patterns and tempi | E003.maxpat

The "playback style" *bpatcher* is another part of the EAMIR SDK. It simply takes the notes of the chord as given by the "chord voicing" *bpatcher*, splits the notes into two lists after the two lowest notes by using another *zl slice 2* object, and bangs each half of the chord, now in two separate messages, according to a rhythmic pattern specified by a *tempo* object within the *bpatcher*.

9. Ctrl+click (Mac) or right click (Windows) the "playback style" *bpatcher* and select *Object>Open Original "EAMIR_playback _style.maxpat"* from the contextual menu

This patch is set to open in Presentation mode. Unlock the patch and put it in Patching mode.

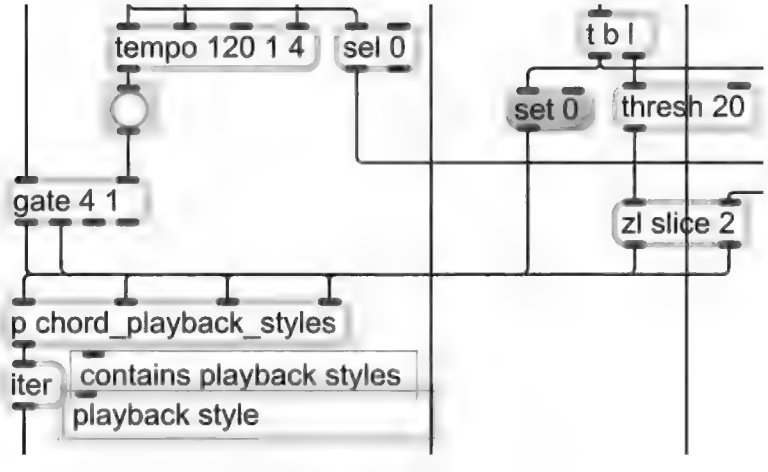


FIGURE 19.6

chord tones are divided and played at different times relative to the tempo | EAMIR_playback _style.maxpat

The notes are received by a *thresh* object and sent to a *zl slice 2* object where they are separated into two lists. Both lists are sent to a subpatcher called *chord_playback_styles* where the lists are triggered in some order similar to the way we created a step sequencer in Chapter 10.

10. Hold ⌘ (Mac) or ctrl (Windows) and double click the subpatcher *chord_playback_styles* to view its contents in a new window

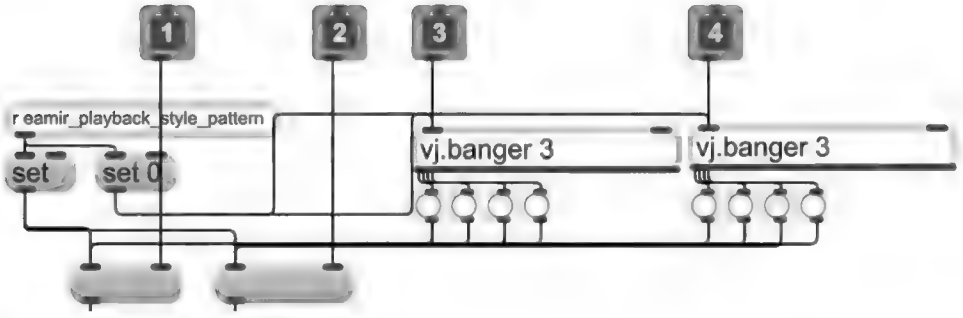


FIGURE 19.7
vj.banger abstraction bangs different chord tones in succession | subpatcher in EAMIR _playback_style.maxpat

In this subpatcher, you can see that both lists are received in the first two inlets and will be used to populate the now-empty *message* boxes. To the right, 4 *buttons* are connected to the *message* boxes in some pattern. When each button is pressed sequentially, both chord lists will be played back in the arranged pattern. A small abstraction called *vj.banger* is used to send a bangs to each of the 4 *buttons* causing the lists to be played back in a pattern.

Close this window along with the subpatcher and the *E003* patch.

The lesson to take away from these 3 different patches is the importance of reusing your work to speed up your process. Each patch functions a little bit differently, but because they were well documented and neatly organized, modifying them to create different patches was easier than rebuilding from scratch. Making your patches modular will save you a lot of time.

E002

The *E004* patch in this collection is basically the same patch as the Interactive White Board (Smart IWB) patch from the EAMIR SDK, so we'll skip to the *E002* patch, which is quite different from the rest. This patch began as an interactive music system I developed with my brother Dan called *automata* (Manzo & Manzo, 2009).

1. Open the file *E002.maxpat* from within the folder *E002* located in the *Chapter 19 Examples* folder

Automata was designed as an installation in which people walking through a gallery would see the main window of the software running on a large computer screen. The screen prompts users to send a text message to *automata @vjmanzo.com* from their cell phones. The text message sent to the e-mail address would have the sender's phone number attached to it (5552229575@phone company.com), numbers that I wanted to let the passersby in the gallery use to make music. In this way, for instance, the number 5 from the phone number

would play scale degree 5 of some user selected mode. What could be more personal than composing music using your own phone number as the pitch material?

Through the programming language PHP, my brother developed code to take the sender's phone number from the text message in the inbox of the e-mail account and set the sender's phone number as the title for a webpage on our server. Using an object in Max called *jweb*, which allows you to view a webpage within Max, I was then able to extract just the numbers from the user's cell phone e-mail address, stripping off the @phonecompany.com part. From there, the user could manipulate his or her phone number in a variety of musical ways using accessible controls to change tempo, timbre, harmony, and more, even with no prior experiences with music.

Automata was later modified for use in a research project I did involving composition and performance with high school students who were not involved in their school's music program (Manzo, 2010). In this modified version, *E002*, the user enters meaningful numbers (locker combination, pet's birthday, etc.) manually by using the onscreen number pad.

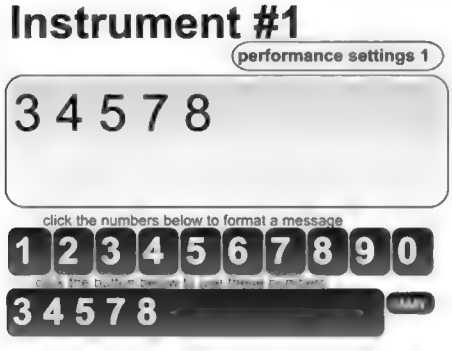


FIGURE 19.8

entering numbers into the E002 patch by clicking message boxes | E002 .maxpat

2. Click some numbers in the first row of numbers and notice that they populate the *message* box below
3. When you have finished entering numbers, click the newly populated *message* box to set the numbers in the light gray area above

There are 4 of these modules; this means that users can have up to 4 different sets of numbers, which will be used as pitch material, to compose and perform with. To perform with the numbers

4. Click the *message* box containing the text *performance settings 1*

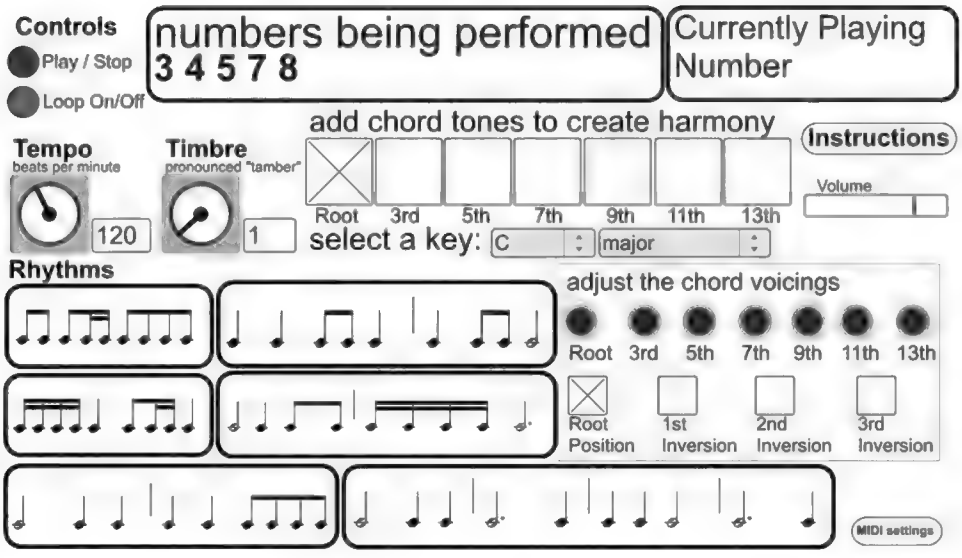


FIGURE 19.9

performance interface for E002 | E002.maxpat

The window that opens contains the user-selected numbers in its queue. When the user clicks *play*, each number in the queue will be played back at the specified tempo, timbre, key, volume, and so on in one of the rhythmic patterns displayed. When this research project was conducted in spring 2010, 4 small touch-screen USB monitors were used so that each window like this one would fit on a single touch-screen monitor, allowing the students to control the musical variables by touching different parts of the screen.[5]

5. Take a few moments and play around with all the settings (students in the research project were given one hour without assistance or interruption and were simply told to "explore")

This control part of the patch is not a subpatcher but another Max patch, which we can open separately from the main patch.

6. Close both patcher windows
7. Open the file *automata_sequence_looper.maxpat* from within the folder *E002* located in the *Chapter 19 Examples* folder
8. Unlock the patch and exit Presentation mode (you may wish to make the patch window larger to see all of the objects)

This patch is actually not much different from the chord progression patch we just discussed. In both cases, single digit numbers are mapped to diatonic scale degrees.[6] The controls are the same things you've seen in other patches, just simply larger, more accessible, colorful, and labeled appropriately for the

5. For more information on *automata* and the documented and videoed results of this research project as well as stand-alone applications of *automata,* visit www.vjmanzo.com/automata.

6. In this patch, the number 9 maps to scale degree 2, and the number 0 maps to scale degree 3.

intended users. The only really different thing in this patch is the predetermined rhythmic patterns used to play back the user's numbers.

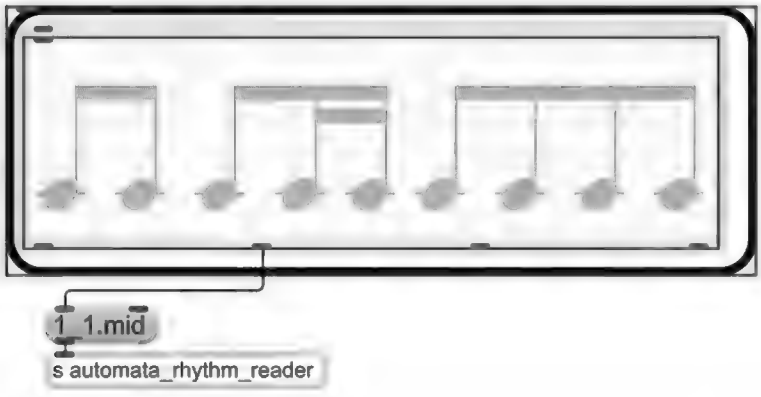


FIGURE 19.10

clicking image loads a MIDI file by which a rhythm is imposed on the pitches | automata _sequence_looper.maxpat

Beneath each image of a rhythmic pattern (loaded by *fpic* from the *E002* folder), there is a *message* box referring to a MIDI file located in the *E002* folder. When the user clicks on the rhythmic pattern picture, a *ubutton* object sends a bang to the *message* box sending the name of a MIDI file to the *s automata _rhythm_reader* object which is received by the *r automata_rhythm_reader* object on the left side of the patch above the *seq* object.

Each MIDI file contains a small recording of a single pitch playing at the same rhythmic value depicted in the *fpic* image. The MIDI data are sent from *seq* to the *midiparse* where the pitch and velocity values are *unpack*ed into *stripnote*. *Stripnote* was used so that "note on" messages would send a bang at the pictured rhythm. If we didn't strip the velocity off messages with *stripnote*, the bangs would occur when the notes were "on" and "off" and the rhythm would not be played as pictured.

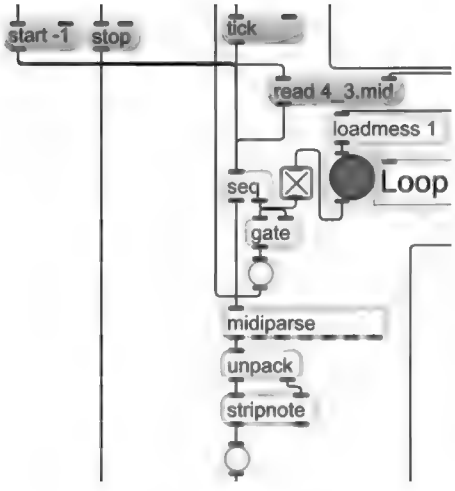


FIGURE 19.11

MIDI files imposed rhythms at specified tempo on the pitches | automata_sequence _looper.maxpat

From *stripnote*, a bang is sent to a *button* each time the pitch in the MIDI file recording is played back. These bangs are used to scroll through each note in the sequence just as we used the bangs from the *tempo* object to scroll through each chord in the progression in the previous example.

Since the patch uses the *tick* option for *seq*, the tempo for the playback of the MIDI file can be altered as well as the other musical variables. Each separate playback module (4 total) is on a separate MIDI channel so none of the notes will interfere with any other.

Even though the numbers for this patch are entered using onscreen controls, the PHP "cell number grabbing" portion of this patch is still available in a subpatcher in the main patch called *gets_number_from_PHP_database*. To get it working, you will have to supply the URL for your own PHP mail server configured to store cell numbers in a database and that can be a real headache. This part of the patch is not discussed for a variety of reasons since it is beyond the scope of Max. The web is full of free resources for understanding PHP programming if you have the desire to learn it.

New Objects Learned:

- *togedge*
- *jweb*

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a patch that would allow someone who knows nothing about music to do something musical with a limited number of controls. Your patch must be "foolproof" and ready to do whatever it does as soon as the patch opens. The controls must be accessible and the instructions clear. Consider designing this patch for use in an installation such as a gallery or a lobby during the intermission of a concert in which a user may have only a limited amount of time to play with the patch (a minute or two) before leaving. Choose your timbres and consider using a non-Max software synthesizer in your design. Feel free to incorporate video and Jitter objects as well. You may also want to use the *jweb* object to get some data from the web. Remember the first sentence, though, as this is your primary objective.

20

Compositions and Perception Tools

In this chapter, we will examine some ways to interact with audio processing objects in formal compositions. Examples of traditional instrumentalists interacting with Max patches in concert performances are common.[1] In the interest of copyright availability, we will examine a composition of mine for E♭ clarinet and computer (a Max patch). The remaining example patches in this chapter will deal with audio processing as it relates to hearing and some aspects of perception.

Composition for E♭ Clarinet and Computer

In this composition, *discourse,* the clarinetist plays from a score while the Max patch "listens" to the performer (using a microphone) and processes the clarinet sound in predetermined ways. The Max patch follows a time-based "score" of its own for performing the effects on the clarinet sound and, thus, processes the audio signal the same way each time the piece is performed. Our purpose in exploring this patch has less to do with the effects that are used or any aesthetic you get from the piece than with the implementation of a usable timeline that both the clarinetist and the computer can perform to.

1. Great examples exist by composers such as Cort Lippe, Luke Dubois, and Jonny Greenwood of Radiohead, just to name a few.

1. Open the file *discourse.maxpat* from within the folder *discourse* located in the *Chapter 20 Examples* folder

When the space bar is pressed, a *clocker* within the patch will begin triggering the events in the Max patch; this is like the score for the computer. These events assume that since the user has pressed the space bar, the patch can expect to hear the notes of the score played back at tempo to coincide with the different audio processing taking place within the patch. Unless you happen to have your E♭ clarinet handy (a PDF of the score is also available in the *discourse* folder), we will use a demo sound file of a synthesized clarinet playing[2] this piece in lieu of actually performing it. This will give us a sense of what the piece would sound like if we were to perform the clarinet part live.

2. Check the *toggle* labeled *Play Demo*
3. Ensure that the *ezadc~* and *ezdac~* objects are on (red) then press the space bar to begin playing the synthesized clarinet part as an emulation of an actual performance (you may wish to open the PDF file of the score located in the folder to follow along)

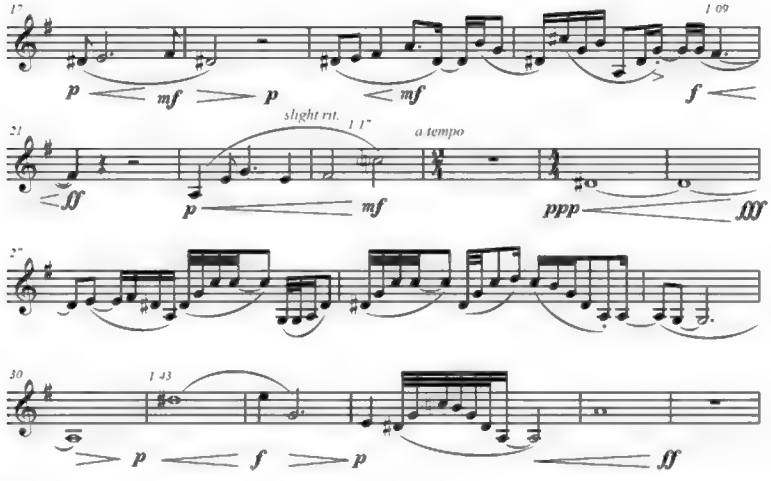


FIGURE 20.1
excerpt from discourse (2009) score with time markings indicated

Notice that the clarinet part has time markings in the score. If the performer plays the piece at the specified tempo, the time markings in the score will line up with the timer within the Max patch. The time markings in the score help to ensure that the performer is playing the part of the score that corresponds to the processes performed on the clarinet's audio signal provided by the Max patch. Time markings in the score are also less strict than asking the performer to play with a metronome/click track, which is also useful, but it tends to perturb some classical musicians.

2. For this purpose, a MIDI file was synthesized with a clarinet sample and rendered as the file *testing.wav*.

4. Continue listening to the piece for at least 2 or 3 minutes until you have heard a number of the effects chime in. You can turn the *ezdac~* off at any time once you have the basic concept of how the piece works (even if you're not crazy about the actual composition)
5. Unlock the patch and exit Presentation mode (you may wish to make the patch window larger to see all the objects)

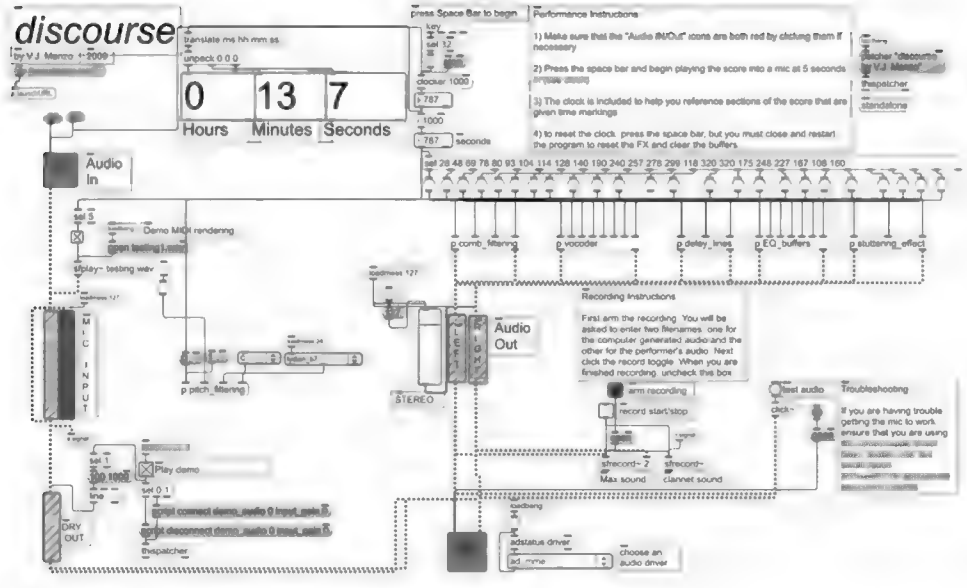


FIGURE 20.2

discourse patch contents| discourse.maxpat

Though the patch may seem a bit involved, it's actually quite simple. Five subpatchers of effects have been implemented similar to the delay and EQ effects we implemented in Chapter 15. For each of these effects, there are several preset values that utilize the same effect in different ways (different EQ settings, delay times, etc.). The Max patch has a *clocker* object that begins counting in seconds when the space bar is pressed. When the *clocker* value reaches certain times, a *sel* object is used to send the audio signal from the microphone to the different effects in the patch.[3] The *sel* object also triggers the various presets and changes in volume and panning. That is essentially the entire piece.

The novelty of the piece is simply in the combination of the clarinet part and the computer part. Since the computer part is driven solely by what is received from the microphone (*ezadc~*), it will do the same processes even if you used a kazoo instead of a clarinet. This, of course, would alter the way the computer part sounds just as if the clarinetist played a few wrong notes and the computer processed those wrong notes. In any case, the computer consistently follows the same score of processes each time the piece is performed.

3. A *timepoint* object could also be used in conjunction with the *transport* object to provide similar control over the effects, as we discussed in Chapter 10.

Similarly, a user could use the automation controls for audio effects inside a DAW and perform the piece that way. However, among other great reasons to use Max instead of a DAW, creating the effects in Max gives you the freedom to build a stand-alone application of the program and easily distribute it to performers along with the score.

One of the more interesting parts of the patch is the part that sends *script* messages to the *thispatcher* object to literally disconnect the patch cord from the *sfplay~* object playing the demo clarinet file to the *gain~* object that amplifies the signal. In the *Inspector* for both of these objects is a field called *Scripting Name*. The *sfplay~* object has been given the *Scripting Name demo_audio* while the *gain~* beneath it has been given the name *input_gain*.

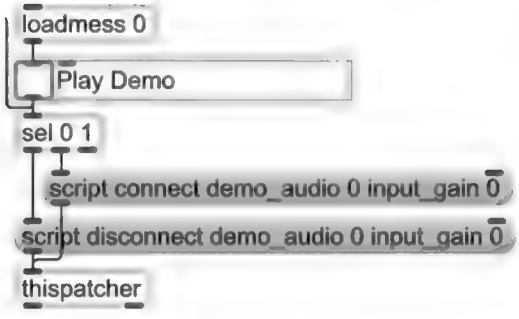


FIGURE 20.3
scripting used to disconnect patch cords | discourse.maxpat

6. Open the *Inspector* for *sfplay~* and *gain~* to see the name in the *Scripting Name* field

Sending *script* messages to *thispatcher* can allow you to create and modify objects within a patch without actually unlocking the patch and editing objects. One such action is used here to either connect or disconnect the first *demo_audio* outlet (*0*) to or from the first *input_gain* inlet (*0*). Of course, we could have alternatively used a *gate~* to restrict the flow of audio from the *sfplay~* to the *gain~*, but then you might never know that this *script* option was available.

Different methods of using *script* messages with *thispatcher* are possible, and even greater control is available through the use of the *Javascript* programming language implemented in the *js* and *jsui* objects. There are numbers of free tutorials online for understanding *Javascript*. Of course, the use of *Javascript* is just another reference to the many programming languages that can be incorporated into Max[4] if you happen to already know or are planning to learn one. Close this patch.

4. Third party objects exist to implement the languages LISP (*maxlispj* by Brad Garton http://music.columbia.edu/~brad/maxlispj/), ChucK (*chuck~* by Brad Garton http://music.columbia.edu/~brad/chuck~/), CSound (*csound~* by Davis Pyon, Matt Ingalls, and Victor Lazzarini http://www.davixology.com/csound~.html) as well as the Max SDK, which allows external objects to be programmed in a number of languages such as C and C++.

Another composition called *nil* for guitar and computer is included in the *Chapter 20 Examples* folder. The main file *nil.maxpat* functions similarly to this one but uses MIDI files to trigger events.

Hearing

Audio processes taking place in a patch have more use for educators than simply just composition or performance situations. One of the most important skills a musician can have is listening skills. A discussion of listening, of course, references the sense of hearing. In 2006, I became aware of "the Mosquito" alarm (and later the ringtone) invented by Howard Stapleton in 2005. The device is used in malls and other shopping areas to prohibit teens and young adults from loitering by emitting high frequencies that only they can hear at loud levels. The basic idea of this device and similar ones originates from statistical data showing that most adults suffer some wear and tear on their ears so that by their mid to late twenties, they cannot hear high frequencies, such as *17.4* kHz, that they were able to hear when they were in their early teens. Thus, playing a loud frequency at *17.4* kHz in a crowded mall or street corner would bother only those people who were able to physically hear it: young people. The concept was later turned into a ringtone so that students could hear a high frequency when they were receiving a phone call while their teacher remained unaware of the sound.

I have included, in the *Chapter 20 Examples* folder, a "mosquito" patch that scrolls through different frequencies showing the approximate age that hearing the frequency is likely to become difficult.

1. Open the patch *mosquito.maxpat* from within the *mosquito* folder in the *Chapter 20 Examples* folder
2. Ensure that the *ezdac~* is on (red), press the space bar, and slowly raise the gain to hear an *8* kHz frequency
3. You may press the space bar again to scroll through the next frequency in the *umenu* if you'd like (Note: determining what frequencies you don't hear so well anymore can be depressing)
4. Turn off the *ezdac~*
5. Unlock the patch and exit Presentation mode (you may wish to make the patch window larger to see all the objects)

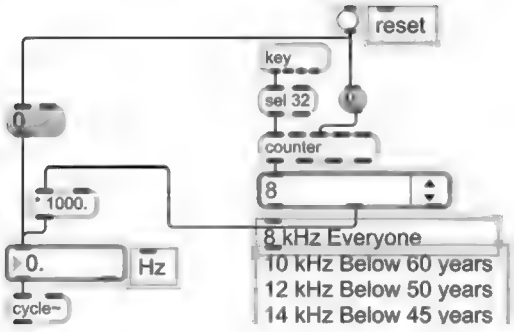


FIGURE 20.4

frequencies stored in a umenu are cycled through and used in this hearing test | mosquito.maxpat

You probably had already figured out how this patch worked before you unlocked it. A bang from the space bar simply counts through the items in the *umenu*. The values in the *umenu* are multiplied by *1000.*, converted into sine waves with *cycle~*, and sent to the *ezdac~*. The frequency and its gain are also displayed in the *spectroscope~* object. This patch is pretty simple, but it's useful for what it does. Close this patch.

Perception

"Arguably, listeners must establish the key of a piece within the first few bars (if not notes) if they are to code and store musical material appropriately" (Sloboda, 2005, p. 129). In studies concerning how individuals perceive harmony, researchers may use models in their experiments that avoid complications associated with pitch height. Pitch height is the quality of harmony by which voice leading and inversion may influence the way someone perceives harmony or harmonic direction[5] (Deutsch, Moore, & Dolson, 1984). Preferably, pitch height should be confounded, leaving only pure harmony for the participant to hear and use to make judgments (Krumhansl & Kessler, 1982; Deutsch & Boulanger, 1984).

Early studies in the perceptual organization of harmony implemented models to confound pitch height based on the so-called *Shepard Tones* named after Roger Shepard (Shepard, 1964).

1. Open the file *shepard.maxpat* inside the *shepard* folder within the *Chapter 20 Examples* folder
2. Turn on the *ezdac~*
3. Click on the *toggle* labeled *Start Progression* and slowly raise the *gain~* slider

Shepard is credited with creating a model similar to Escher's ever-ascending staircase in which the listener perceives a pitch to be continually rising in an infinite glissando (Shepard, 1964). The illusion of the "ever-ascending major scale" is better experienced while listening through headphones as would be the case in research experiments.

4. Turn off the *ezdac~*
5. Double click one of the objects called *shepard_synth* to open a window displaying the patch's contents

5. In other words, a high and loud F♯ in a D dominant 7th chord in the key of G major may influence a listener to hear the chord as resolving to a G major chord because of the way the chord was voiced.

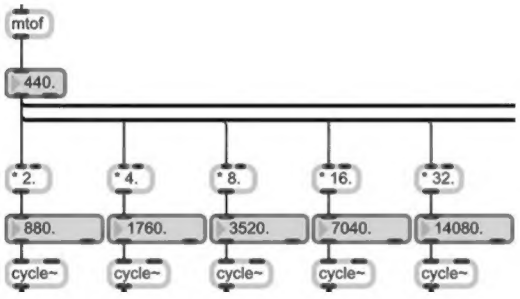


FIGURE 20.5

like the harmonic series but only octave multiples are present | shepard .maxpat

In *Shepard Tones*, the only harmonics are multiples of the fundamental frequency that are powers of *2*. In other words, it's like the harmonic series patch we made in Chapter 12, but we limit the frequencies so that we are left with only octave doublings of the fundamental frequency at each audible octave spanning the full range of hearing (20Hz–20kHtz), a timbre that sounds much like an organ. By having every audible octave frequency present for each tone in a chord, the effects of inversion and chord voicing for the chord are lost. A researcher could then make judgments about the participant's perception of harmony knowing that pitch height was eliminated to some extent. Close this window.

6. Double click one of the objects called *shepard_filter* to open a window displaying the patch's contents

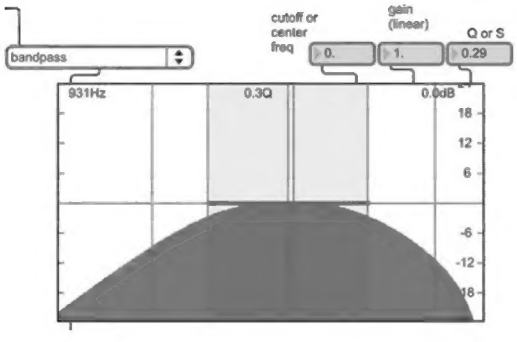


FIGURE 20.6

filtergraph~ used to shape contour | shepard _filter.maxpat

A filter is often used in perception studies because even though all the frequencies may be at the same level, they may be perceived at different levels of "loudness" by the ear. In this patch, and in many early experiements, a bell-curve (Gaussian filter) is used to taper off the levels of the frequencies at the extremes of the signal. Notice that this is the same type of *filtergraph~* object we used in Chapter 15. Close this window and close this patch.

The idea of "Equal Loudness" contours, such as the well-known *Fletcher-Munson* curves (Fletcher & Munson, 1933) and the latter standard *ISO226* curve (International Organization for Standardization), resulted from multiple tests in perception of frequencies across the spectrum. As a result, the term *phon* is used as a unit of perceived loudness.

1. Open the file *iso226.maxpat* inside the *iso226* folder within the *Chapter 20 Examples* folder
2. Ensure that the *ezdac~* is on (red) and click on the *toggle* labeled *Start Progression*
3. Slowly raise the *slider* on the left to increase the *phon* level

Notice that this patch, while similar to the *Shepard* patch, sounds different since it is based on a different standard of equal loudness perception. The patch is identical to the *shepard* patch we looked at earlier except that it contains an

implementation of the ISO226 contour in place of the Shepard tones. Building models for aural perception and testing can be done using these models as a template.

Conclusion

As you continue to work with Max, remember the techniques we've discussed including conceptualizing the programming task, dividing the programming task into smaller tasks, and making your patches modular so that you can reuse working sections of code. Remember also that the Max language can be expanded with the use of abstractions and externals.

As music educators, we are always open to new tools for illustrations, demonstrations, performance and composition activities, as well as research and testing, so while Max can do many great things, it is important to think of the language in terms of how it can serve your objectives and how you will use it. Such tools may not only enhance and serve your existing teaching objectives but can change the pedagogies used in music education.

For the future, you may find it helpful to explore the *example* folder bundled with Max in the Max application folder. These patches contain many great examples of working patches you can adapt into new or existing patches. There are additional projects you can analyze in the "Extras" folder bundled with the chapter examples you've downloaded. Remember, while programming, to try to visualize each step in the process, and divide the task up into smaller modular parts. Consult the Help files, Reference pages, and Tutorials often, and remember that reading through the Cycling '74 forums can be a helpful resource to find answers when you get stuck. Happy Maxing!

New Objects Learned:

- *spectroscope~*

On Your Own:

- Read the Help file for each of the new objects introduced in this chapter.
- Create a composition patch that changes effects for the incoming audio signal over time. The composition can be for a traditional acoustic instrument such as a singer or a spoken voice, or an electric one like an electric guitar. Your patch effects should show some signs of thought and relation. You may also write a notated score for the human performer to follow.
- Examine the Lapp program in the *Chapter 20 Examples* folder

References

Cope, D. (1991). *Computers and Musical Style.* Madison, Wisc.: A-R Editions.

Cope, D. (1996). *Experiments in Musical Intelligence.* Madison, Wisc.: A-R Editions.

Cope, D. (2000). *The Algorithmic Composer.* Madison, Wisc.: A-R Editions.

Deutsch, D., & Boulanger, R. C. (1984). Octave equivalence and the immediate recall of pitch sequences. *Music Perception* (2), 40–51.

Deutsch, D., Moore, F. R., & Dolson, M. (1984). Pitch classes differ with respect to height. *Music Perception* (2), 265–271.

Fletcher, H., & Munson, W. A. (1933). Loudness, its definition, measurement and calculation. *Journal of the Acoustical Society of America* (5), 82–108.

Garton, B. (2009). *maxlispj.* Retrieved 2010, from maxlispj/: http://music.columbia.edu/~brad/maxlispj/.

Green, L. (2008). *Music, Informal Learning and the School: A New Classroom Pedagogy.* Surrey, England: Ashgate.

International Organization for Standardization. (n.d.). *ISO 226:2003.* Retrieved March 10, 2009, from International Organization for Standardization: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=34222.

Jehan, T. (n.d.). Tristan Jehan. Retrieved 2010, from Tristan Jehan: http://web.media.mit.edu/~tristan/.

Krumhansl, C. L., & Kessler, E. J. (1982). Tracing the dynamic changes in perceived tonal organization. *Psychological Review*, 89, 334–368.

LoopBe1. (n.d.). Retrieved 2010, from LoopBe1: http://nerds.de/en/loopbe1.html.

Machover, T. 1999. Technology and the future of music. Interview by F. Oteri, August 18, 1999. *NewMusicBox* 6. http://www.newmusicbox.org.

Manzo, V. J. (2010). *Computer-aided Composition with High-school Non-music Students.* Philadelphia: TI:ME Action Research.

Manzo, V. J., & Dammers, R. (2010, July). Interactive Music Technology Curriculum Project. Retrieved July 2010, from Interactive Music Technology Curriculum Project: http://www.imtcp.org.

Manzo, V. J., & Manzo, D. (2009, July). automata. Retrieved 2009, from automata: http://www.vjmanzo.com/automata.

Puckette, M. (2007). *Theory and Techniques of Electronic Music.* Singapore: World Scientific Press.

Puckette, M. (n.d.). Max/MSP Externals. Retrieved 2010, from Max/MSP Externals: http://crca.ucsd.edu/~tapel/software.html.

Rowe, R. (1993). *Interactive Music Systems.* Cambridge, Mass.: MIT Press.

Shepard, R. N. (1964). Circularity in judgments of relative pitch. *Journal of the Acoustical Society of America* (36), 2346–2353.

Sloboda, J. (2005). *Exploring the Musical Mind.* Oxford: Oxford University Press.

Strand, K. (2006). Survey of Indiana music teachers on using composition in the classroom. *Journal of Research in Music Education*, 54 (2), 154–167.

Thomas, R. B. (1970). *MMCP Synthesis: A Structure for Music Education*. Bardonia, N.Y.: Media Materials.

Tiemann, M. (2006, July 13). Buchla 200e.jpg. GNU Free Documentation License, Version 1.2, Creative Commons Attribution-ShareAlike 3.0.

Young, R. W. (1939). Terminology for logarithmic frequency units. *Journal of the Acoustical Society of America*, 11 (1), 134–139.